ゲーム アルゴリズム レシピ

Game Algorithm Recipe

for JavaScript

松浦健一郎・司ゆき 著

秀和システム

ゲーム アルゴリズム レシピ

Game Algorithm Recipe

for JavaScript

松浦健一郎・司ゆき 著

秀和システム

はじめに

色々なゲームを遊んでいると、自分でもゲームを作ってみたくなりませんか。本書は初めてゲームを作る方にも楽しんでいただけるゲームプログラミング解説書を目指しました。図解を多く取り入れて平易な説明を心がけましたので、プログラミングをする方にはもちろん、プログラミングをしない方にもゲームの作り方を分かりやすく学んでいただけると思います。

最近のゲームは複雑で、一見とても作れなさそうに思えますが、実は基本となる仕組みは意外に単純です。本書はこういった基本的な仕組みを、キャラクターの画像やアニメーションを表示するところから始めて、段階的に解説していきます。プレイヤーの入力に応じてキャラクターを動かしたり、敵や障害物との接触を判定したり、スコアを加算したりといった具合に、少しずつゲームを発展させます。

基本だけで終わりではありません。敵を自動的に動かしたり、武器で敵を撃ったり、マップをスクロールさせたり、アイテムでパワーアップしたりといった具合に、ゲームはどんどん本格的になっていきます。物を押して動かしたり、トランポリンで跳んだりといった、よく知られているゲームのアクションを再現する方法も解説します。

終盤では、タイトル画面やコンティニュー画面を追加したり、ネットワークを使ったハイスコアランキング機能を実現したりして、ゲームを完成させます。本書を読み終える頃には、さまざまなゲームを自力で作れるようになっていることでしょう。あとはご自身のアイディアを活かして、オリジナルゲームの制作に没頭してください。

本書はプログラミング言語の種類に関係なくお使いいただけますが、サンプルの制作にはJavaScriptを使いました。JavaScriptのプログラムはWebブラウザがあれば誰でも実行できるので、制作したゲームを多くの人に遊んでもらうのに最適です。開発にも特別な環境は必要ありません。配布もアップデートも簡単ですし、Ajaxを使ったネットワークプログラミングも可能です。さらに、Flashのプログラミング言語であるActionScriptと言語仕様が共通なので、JavaScriptで学んだ知識をFlashプログラミングに活用することもできます。

ゲーム作りは難しくありません。ゲームに興味を持っている方が、本書をきっかけにゲーム制作の世界を心ゆくまで楽しんでいただけるようになれば、筆者にとって至上の喜びです。

2008年冬

松浦 健一郎 / 司 ゆき

目 次

Chapter 10 画面の遷移 309

Chapter 11 ネットワーク 335

Chapter 1

キャラクター

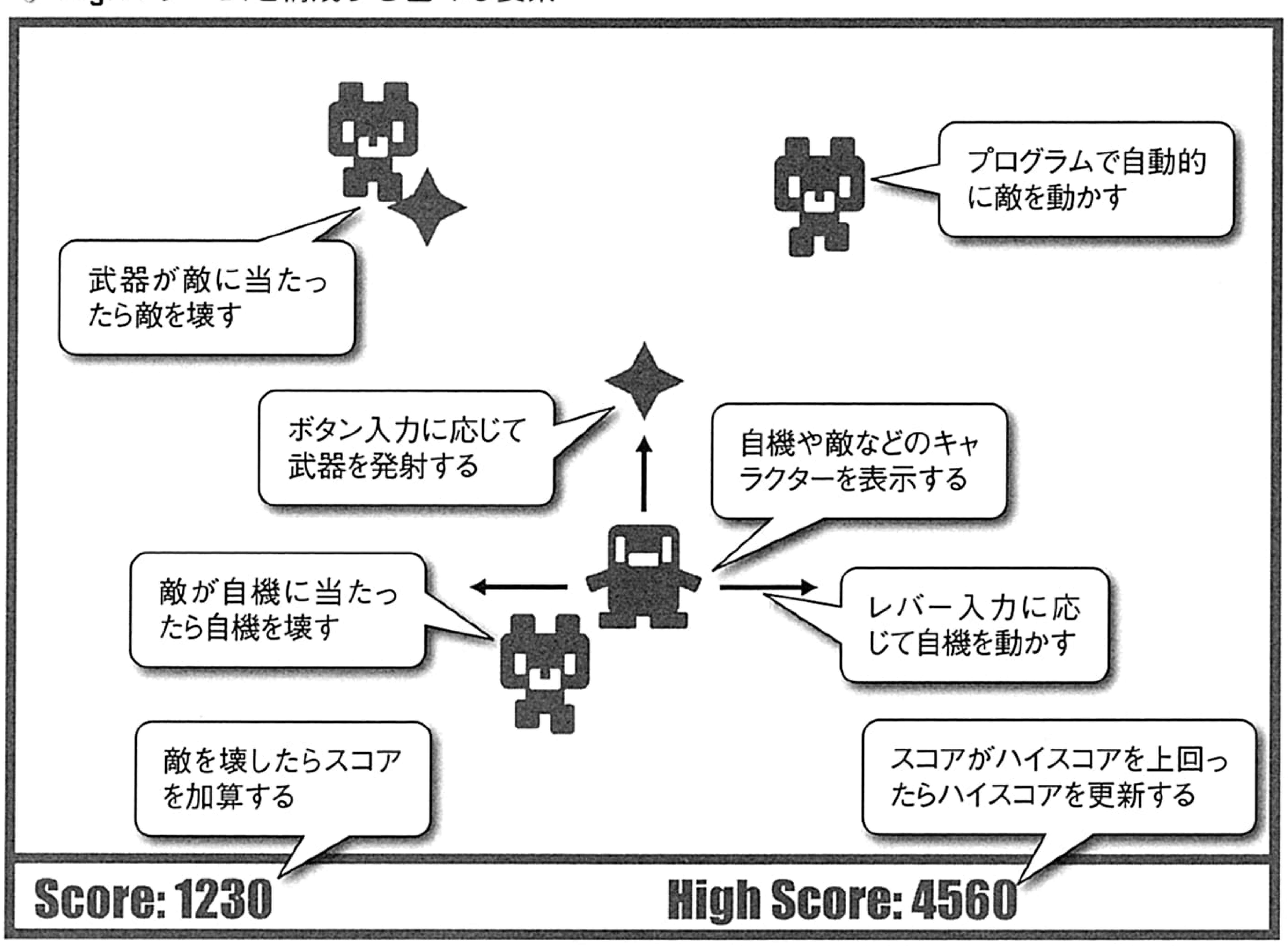
ゲームを作る手順

これからゲームを作ってゆきます。最初に、ゲームの制作手順について考えてみましょう。ゲームは色々な要素から構成されています(Fig.1)。単純なシューティングゲームを例にすると、ざっと思いつくだけでも、

- 自機や敵などのキャラクターを表示する
- レバー入力に応じて自機を動かす
- プログラムで自動的に敵を動かす
- ボタン入力に応じて武器を発射する
- 武器が敵に当たったら敵を壊す
- 敵が自機に当たったら自機を壊す
- 敵を壊したらスコアを加算する
- スコアがハイスコアを上回ったらハイスコアを更新する

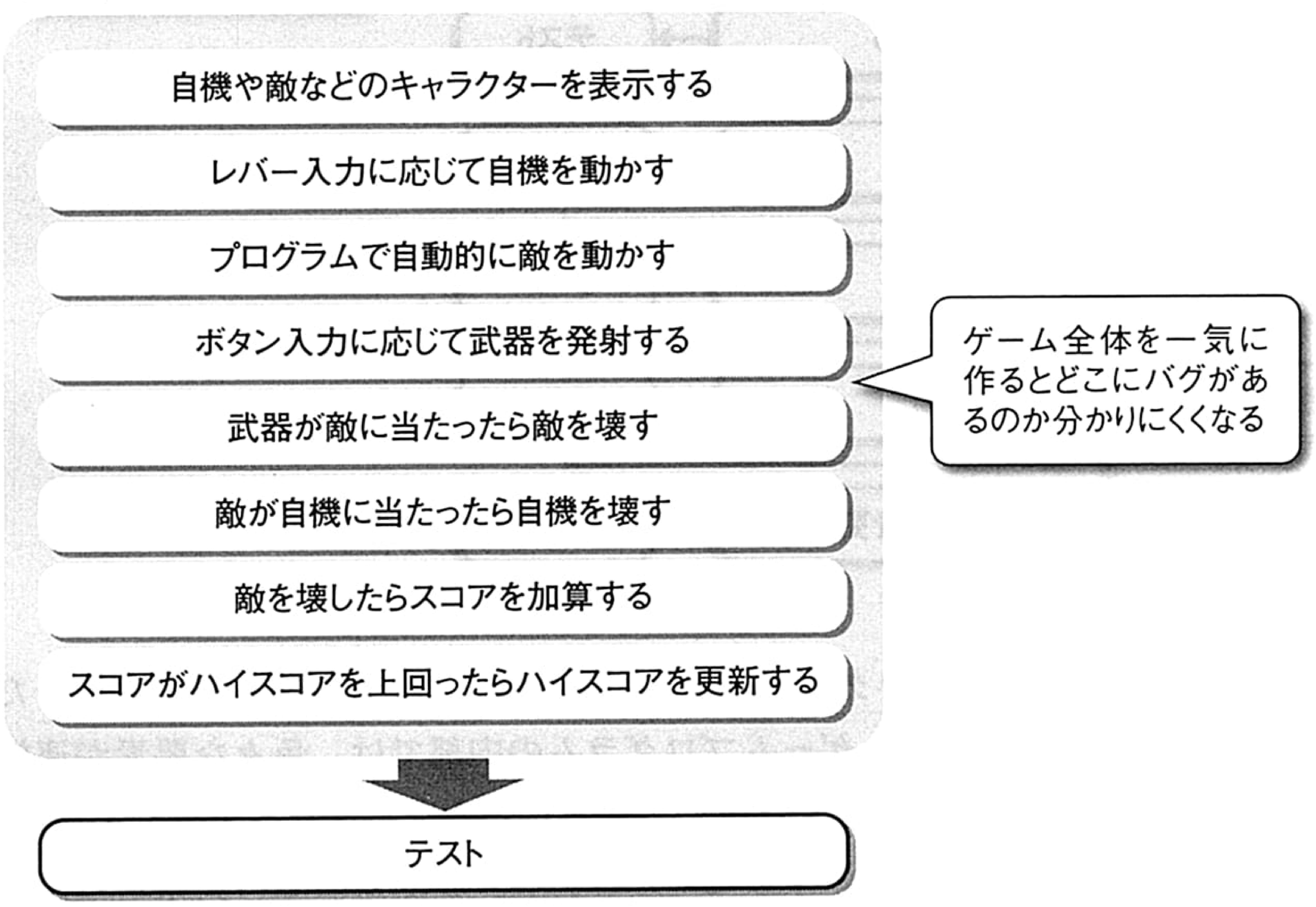
といった具合に、実にさまざまな要素が必要です。

▼ **Fig.1**: ゲームを構成する色々な要素



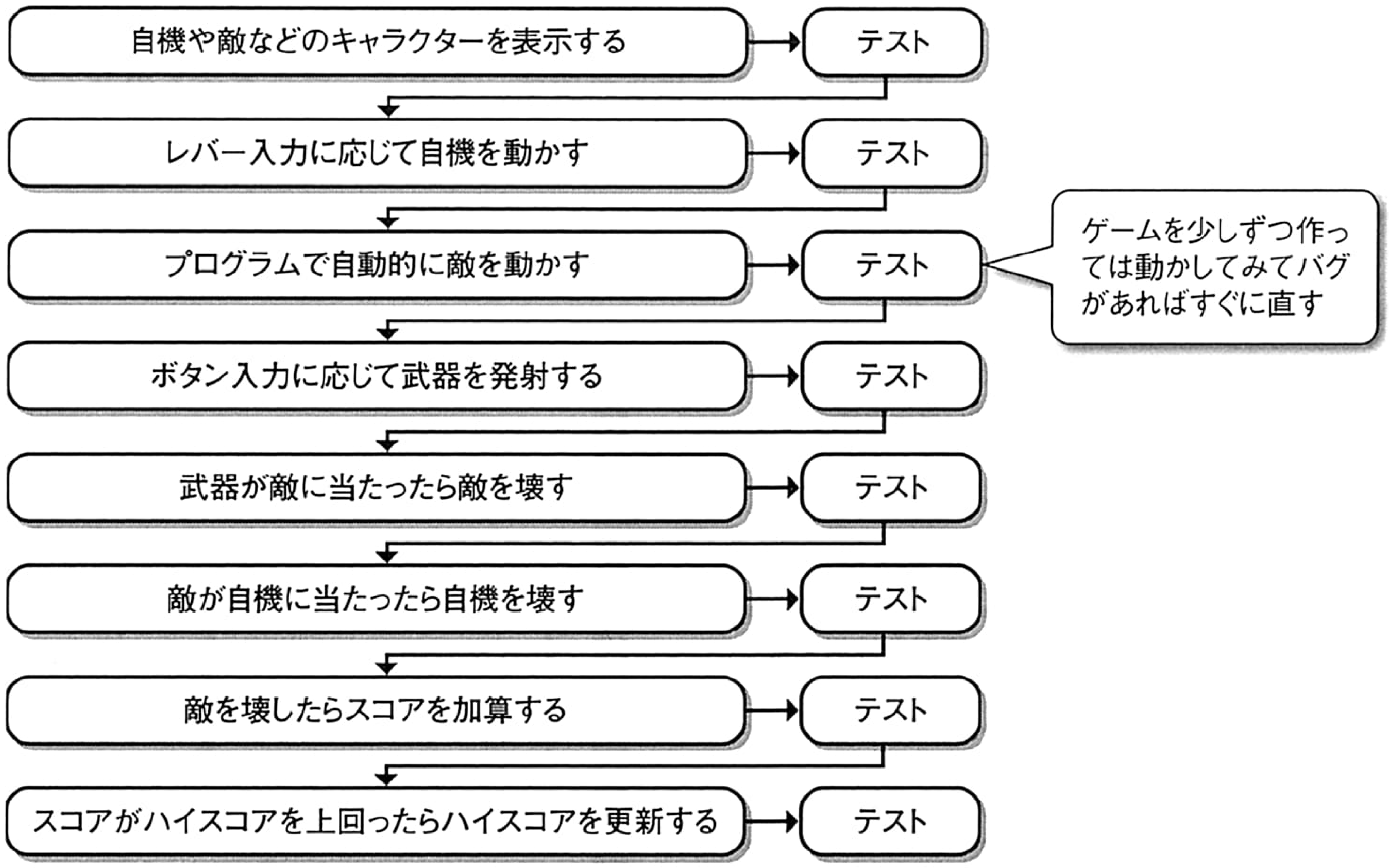
これだけ多くの要素があるので、ゲーム全体を一気に作り上げるのは、なかなか大変です。特に、プログラムにはバグ(プログラムの誤り)がつきものなので、たとえゲームプログラムを一度に書き上げたとしても、正しく動くことはまれです。その上、プログラムのどこにバグがあるのかが分かりにくくなるため、デバッグ(バグを発見して除去すること)に大変な苦労をします(Fig.2)。

▼ **Fig.2**: ゲーム全体を一気に作る



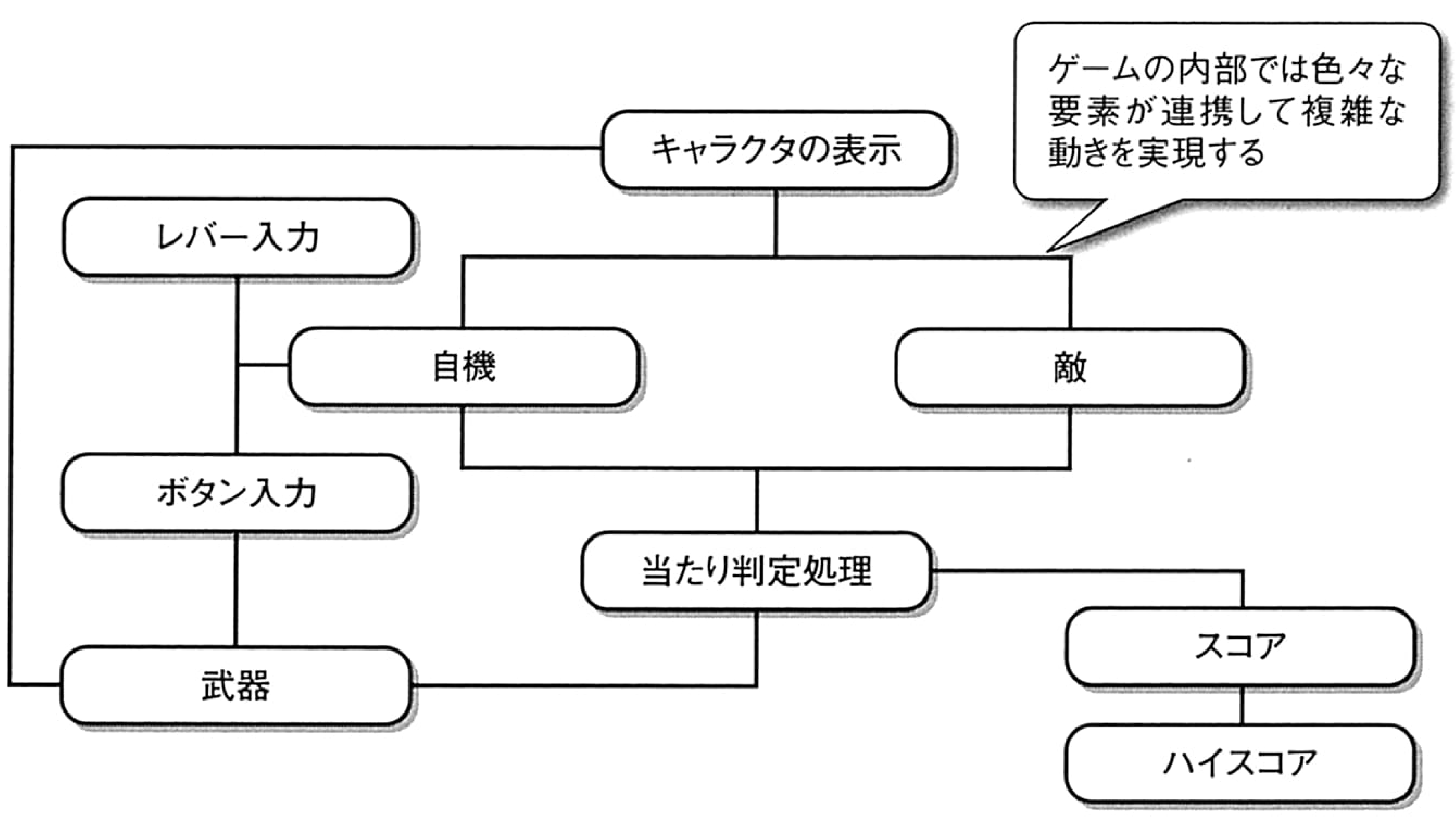
一方、ゲームを少しずつ作れば、デバッグはとても簡単になります(Fig.3)。プログラムを少し書いたら、動かしてみて、バグがあればすぐに直します。正しく動くようになったら、またプログラムを少し書き足して、再び動かし、デバッグを行います。この手順を繰り返して、だんだんゲームを完成させます。

▼ **Fig.3**: ゲームを少しずつ作る



ゲームを少しずつ作るのは、デバッグを楽にするという点で、とてもよい方法です。ただし、制作手順には多少の工夫が必要です。ゲームプログラムの内部では、色々な要素が連携して、ゲーム全体の複雑な動きを実現しています(Fig.4)。こういったプログラムを少しずつ作りつつも、常に動かしてデバッグできる状態にするためには、プログラムのどの要素から作るのか、その要素をどのようにテストするのかを、よく考えておく必要があります。

▼ **Fig.4**: 色々な要素が連携している



最初は、できるだけ基本的で、できるだけ他の部分から独立した要素から着手するとよいでしょう。例えば「キャラクターを表示する」というのは、非常に基本的で、かつ他の部分とは独立してテストできるので、始めに取りかかれる要素です。それに対して、「武器が敵に当たったら敵を壊す」といった要素は、複雑で、他の部分とも絡み合っているので、ある程度制作が進んでから取りかかります。

「キャラクターを表示する」という要素から制作するのには、ゲームにとって不可欠なグラフィックス処理（画面を描画する処理）から取りかかるという点でも、適切な手順です。ゲームには、

- グラフィックス
- サウンド
- 入力
- ネットワーク

といった処理がありますが、このうち最も重要なのがグラフィックスです。サウンド（音）がないゲームや、ネットワーク（通信）がないゲームは珍しくありませんが、グラフィックスがないゲームはほとんどありません。キーボードやマウスからの入力処理は多くのゲームにありますが、グラフィックスに比べると難しい処理ではありませんので、やはりグラフィックス処理から取りかかるのがお勧めです。

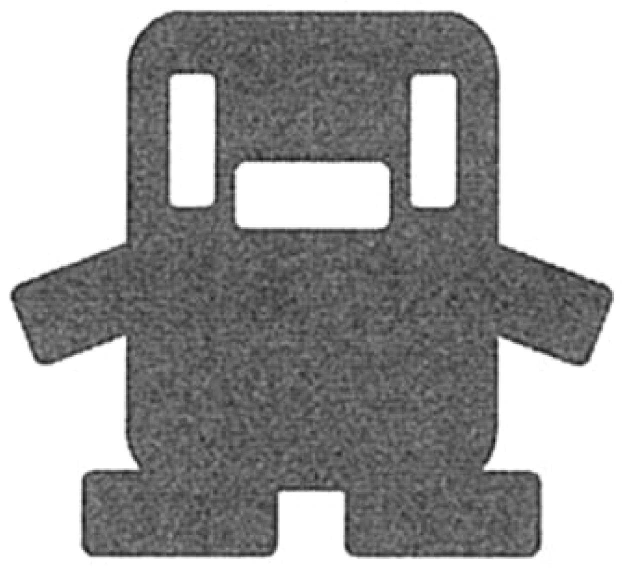
キャラクターの画像

ということで、キャラクターを表示することから、ゲームの制作を始めましょう。キャラクターを表示することができれば、ゲームの具体的なイメージがわきやすくなりますし、開発環境の使い方にも慣れることができます。

まず、キャラクターの画像を用意します。画像は何でも構いませんが、できれば既存のキャラクターよりも、オリジナルのキャラクターがよいでしょう。オリジナルのキャラクターならば、作品が完成したときに、著作権などを気にせずに配布することができます。後で画像を入れ替えるのは簡単なので、最初は何かシンプルなキャラクターを、自分で描いてみるのがお勧めです。

本書では一昔前のアクションゲームをイメージして、こんなキャラクターを描いてみました（Fig.5）。ペンギンのような、あるいはヒヨコのようなキャラクターです。

▼ **Fig.5**: キャラクターの例

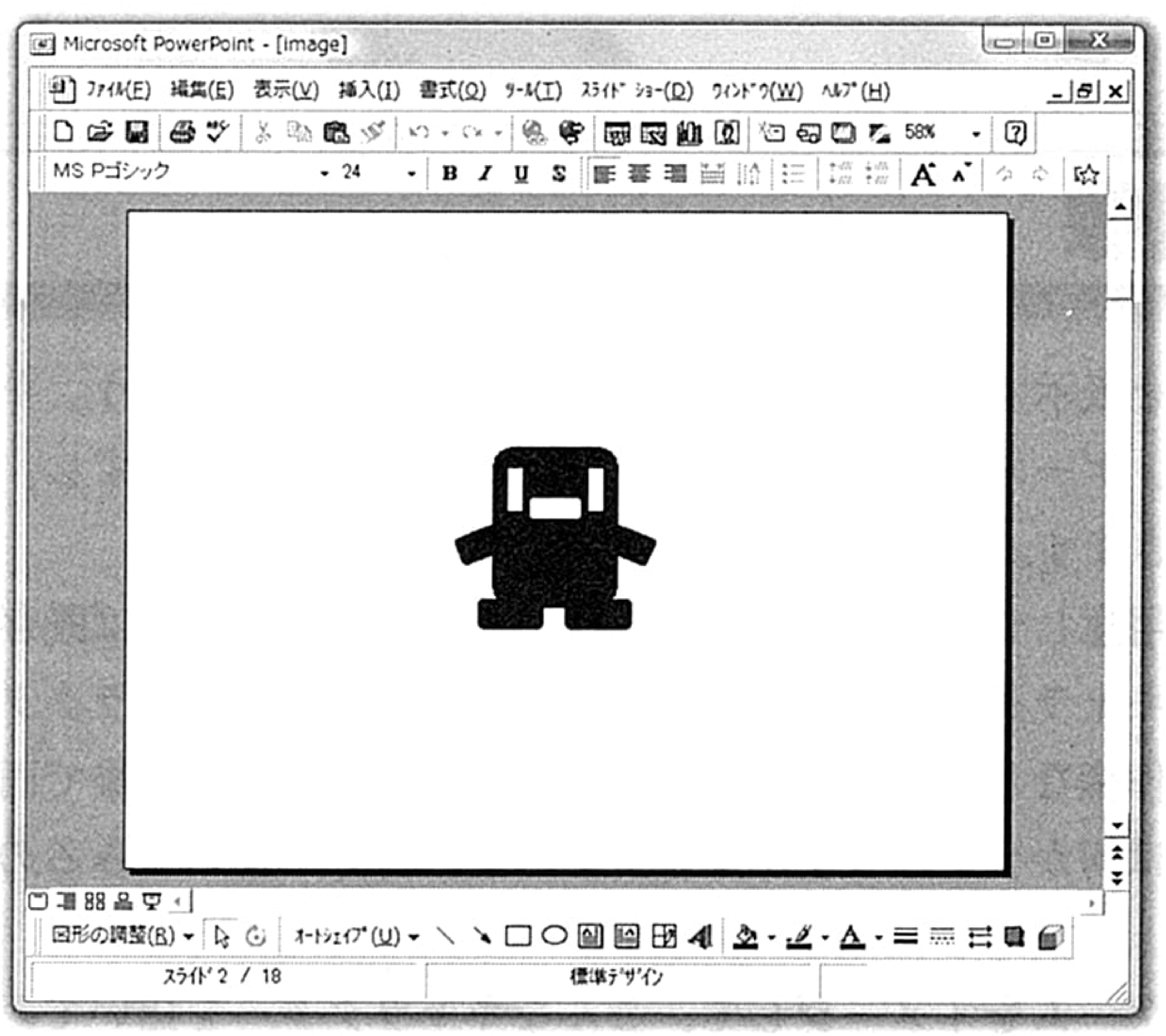


キャラクターを描くのには、どんなツールを使っても構いません。PhotoshopやPainterといった市販の画像ツールはもちろん、フリーソフトウェアを使うこともできますし、Windowsに付属するペイントを使うことも可能です。

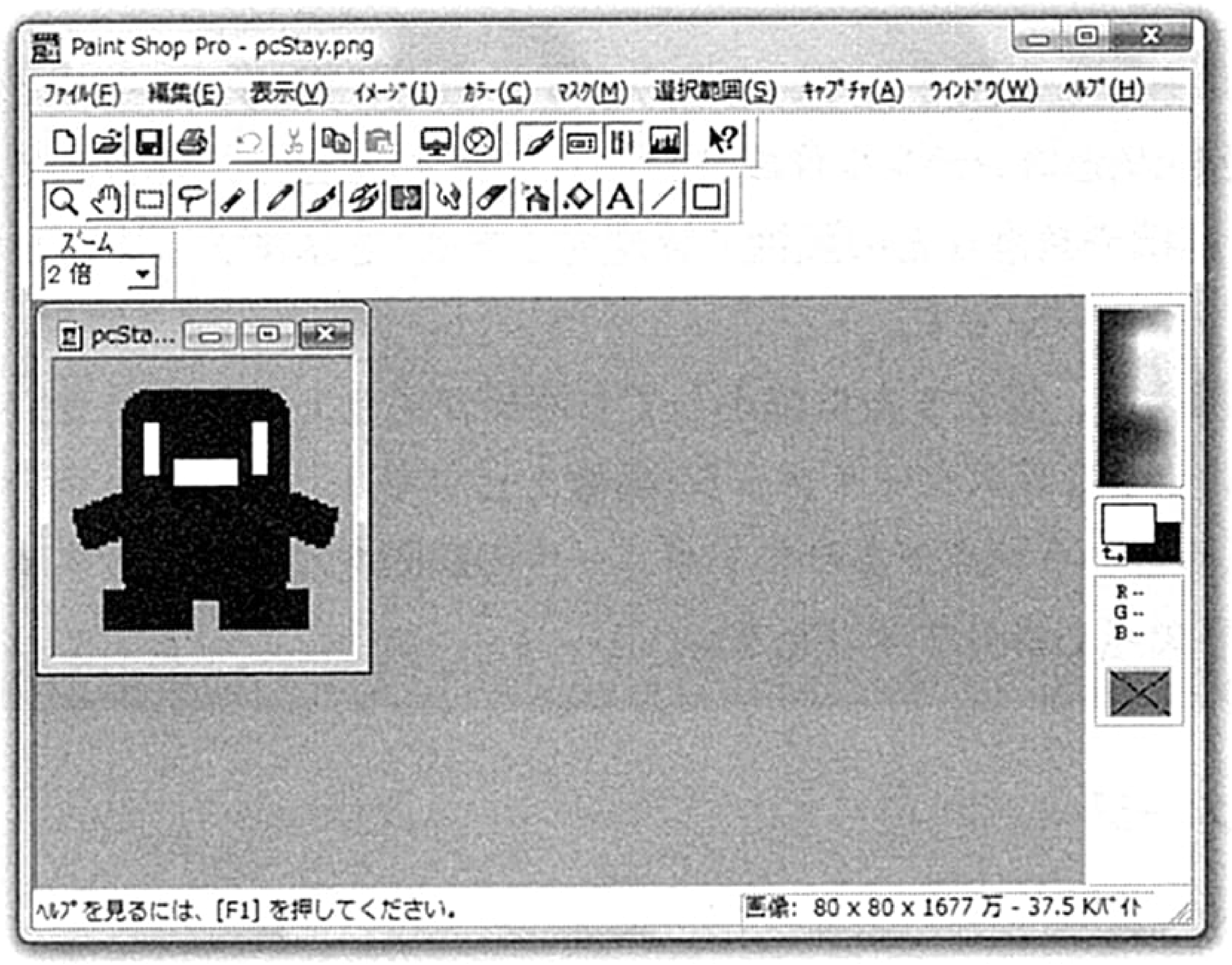
ただし後で説明するように、キャラクター画像の場合には、背景を透明にする必要があります。背景を透明にする機能があるかどうかは、画像ツールによって異なります。例えばWindows付属のペイントには、この機能はありません。本格的にゲームを制作するためには、別の画像ツールを用意しておくとよいでしょう。

ちなみに本書の画像は、Microsoft PowerPointと、Paint Shop Proを使って作成しています。最初にPowerPointを使い、いくつかの図形を組み合わせて、キャラクターを作ります(Fig.6)。次に、作ったキャラクターをPaint Shop Proにコピー＆ペーストして、画像に変換します(Fig.7)。

▼ **Fig.6**: PowerPointでキャラクターの形を作る



▼ **Fig.7**: Paint Shop Proで画像に変換する

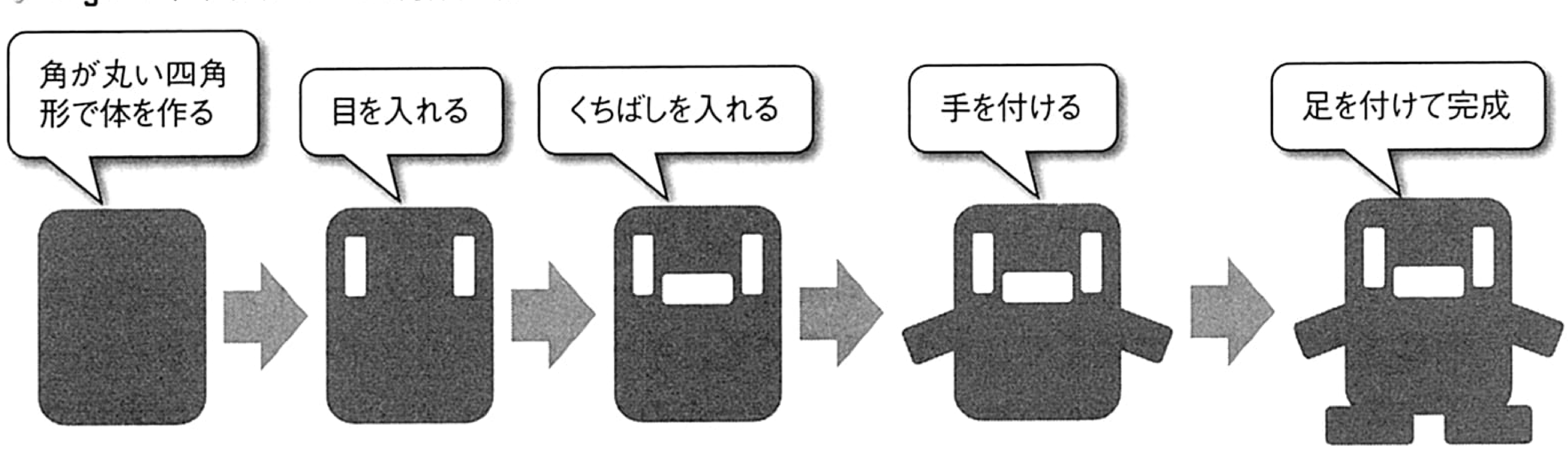


PowerPointは、矩形や円といった図形を組み合わせて形を作る、ドロー系の画像ツールです。一方のPaint Shop Proは、点や線を描いたり、領域を塗りつぶしたりして絵を作る、ペイント系の画像ツールです。

ドローツール(ドロー系の画像ツール)は、図形を素早く作ったり、作った図形を修正したりするのに優れているので、シンプルなキャラクターを作るのには大変便利です。一方、作った図形をゲームで使うためには、PNGやJPEGといった形式の画像に変換する必要があるので、本書ではペイントツール(ペイント系の画像ツール)を組み合わせています。

なお、サンプルのキャラクターは、矩形や角が丸い四角形を並べたり、回転させて配置したりして作りました(Fig.8)。図形の位置や角度を何度でも簡単に編集できるところが、ドローツールのよいところです。

▼ **Fig.8**: キャラクターの制作過程

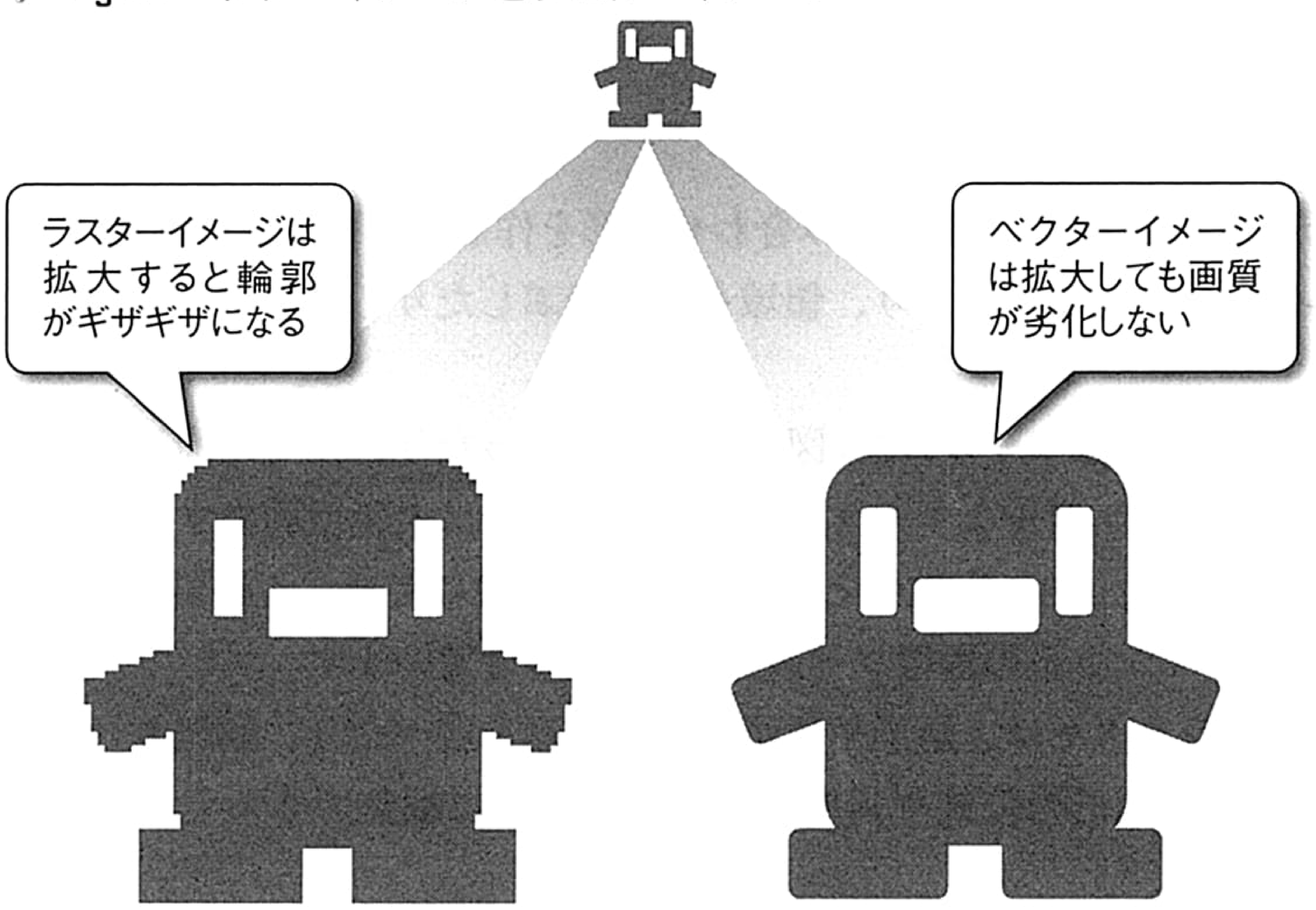


ドローツールとゲーム制作

ドローツールは、これからのゲーム制作ではもっと活用されていきそうです。最近、Webブラウザで遊べるゲームの多くはFlash(フラッシュ:Adobe社のWebコンテンツ作成フレームワーク)を使って作られていますが、このFlashはベクターイメージ(図形を座標や角度などの属性で管理する方式)を基本としています。ドローツールが扱うのは、まさにこのベクターイメージです。

ベクターイメージには、制作のしやすさに加えて、拡大または縮小しても画質が劣化しないという大きな利点があります。これに対して、一般の画像のようなラスターイメージ(画像をピクセルと呼ばれる点の集まりとして管理する方式)には、緻密な絵が作れるという利点がありますが、一定の範囲を超えて拡大すると、ピクセルがギザギザに表示されてしまうという欠点もあります(Fig.9)。

Fig.9: ベクターイメージとラスターイメージ



ブラウザで遊ぶゲームは、軽い気持ちでプレイするゲームが多いためか、緻密なラスターイメージを使ったものだけではなく、ラフなベクターイメージを使ったゲームも歓迎されています。また、ディスプレイの解像度が利用者によって異なるため、画像を拡大・縮小することが多く、画質が劣化しないベクターイメージの長所を活かすことができます。

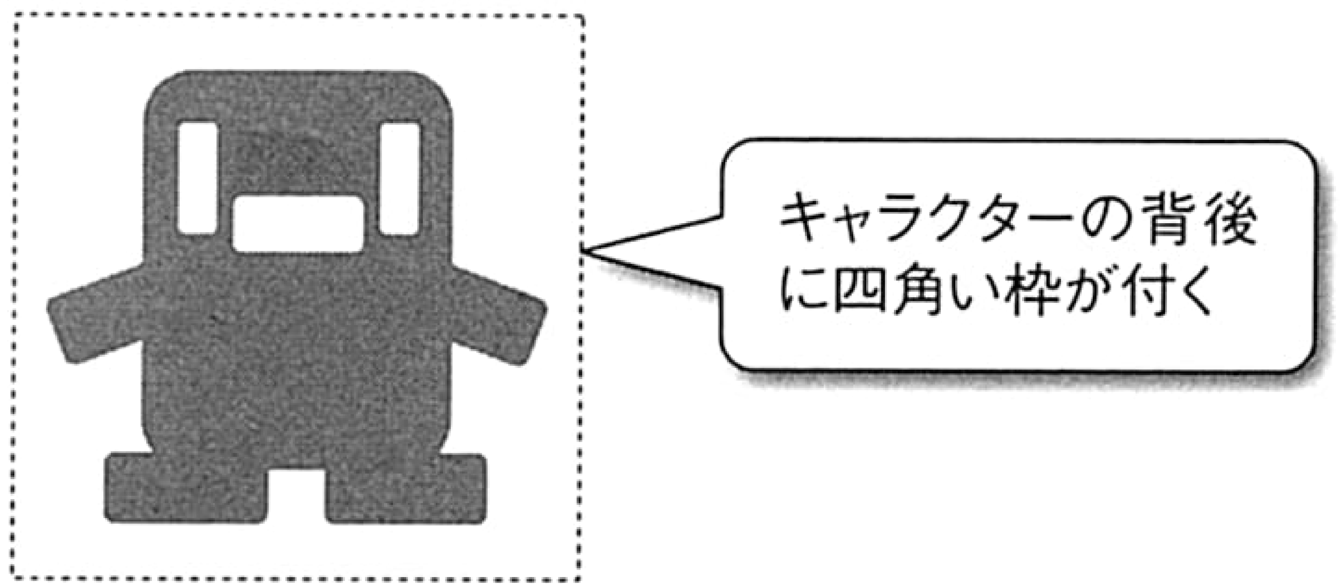
こういった事情から、今後のゲーム制作でドローツールやベクターイメージが活用される場面はますます増えていくでしょう。本書でも、キャラクターのデザインにドローツールを使っています。ただし、JavaScriptではドローツールのベクターイメージを直接には表示できないため、ペイントツールを使って、いったんラスターイメージに変換しています。

SVG(Scalable Vector Graphics)などを使えば、JavaScriptでベクターイメージを扱うこともできます。SVGはXMLを使ってベクターイメージを記述する仕組みです。SVGはブラウザによって対応状況が異なり、SVGデータの編集が可能なツールも限られているため、本書では利用しなかったのですが、将来の発展が楽しみな機能です。

キャラクターの背後を透明にする

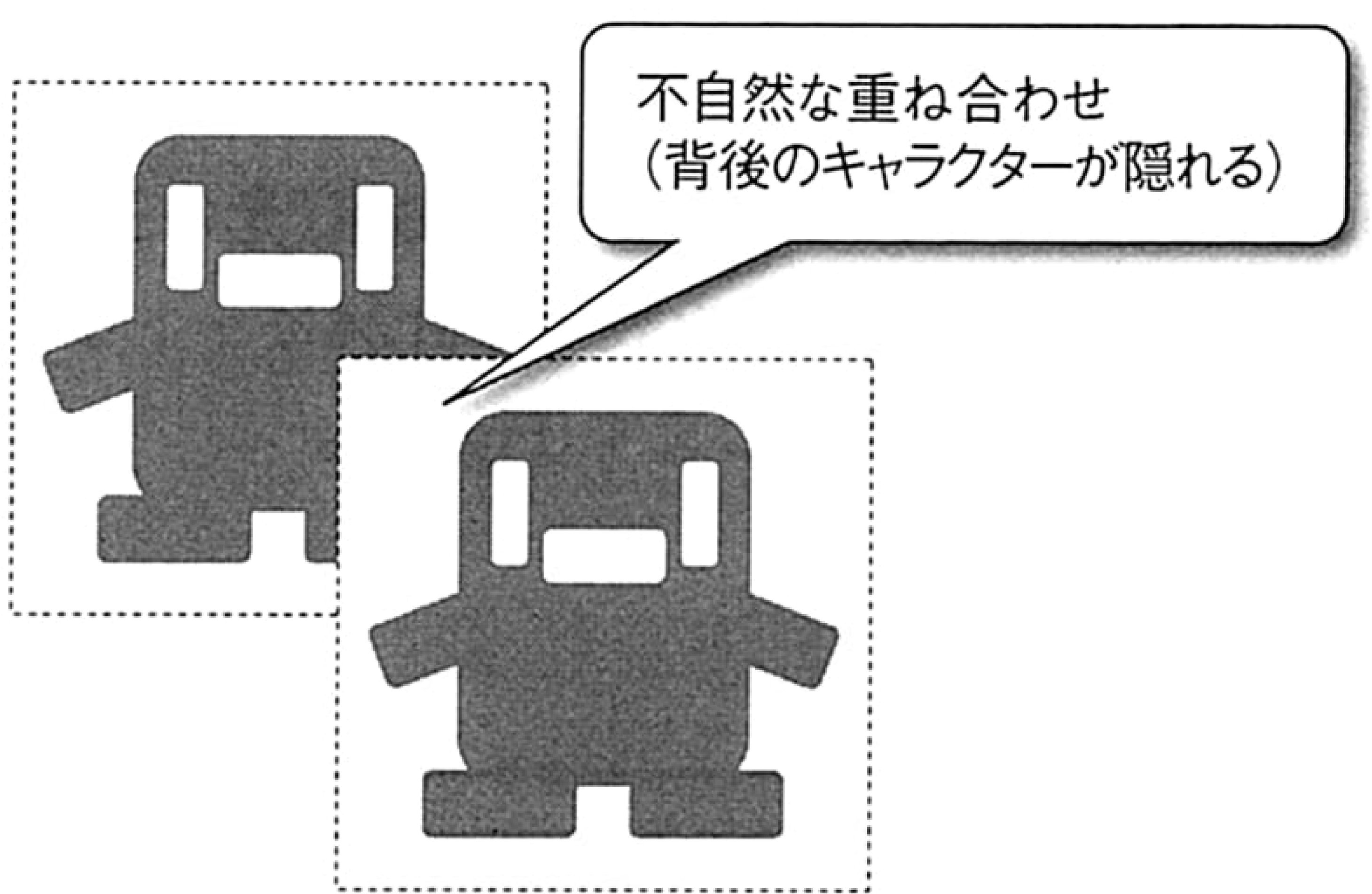
キャラクターの画像を普通に作ると、キャラクターの背後に四角い枠が付きます(Fig.11)。キャラクターは色々な形をしていますが、画像全体としては矩形だからです。

▼ **Fig.10**: 背後の四角い枠

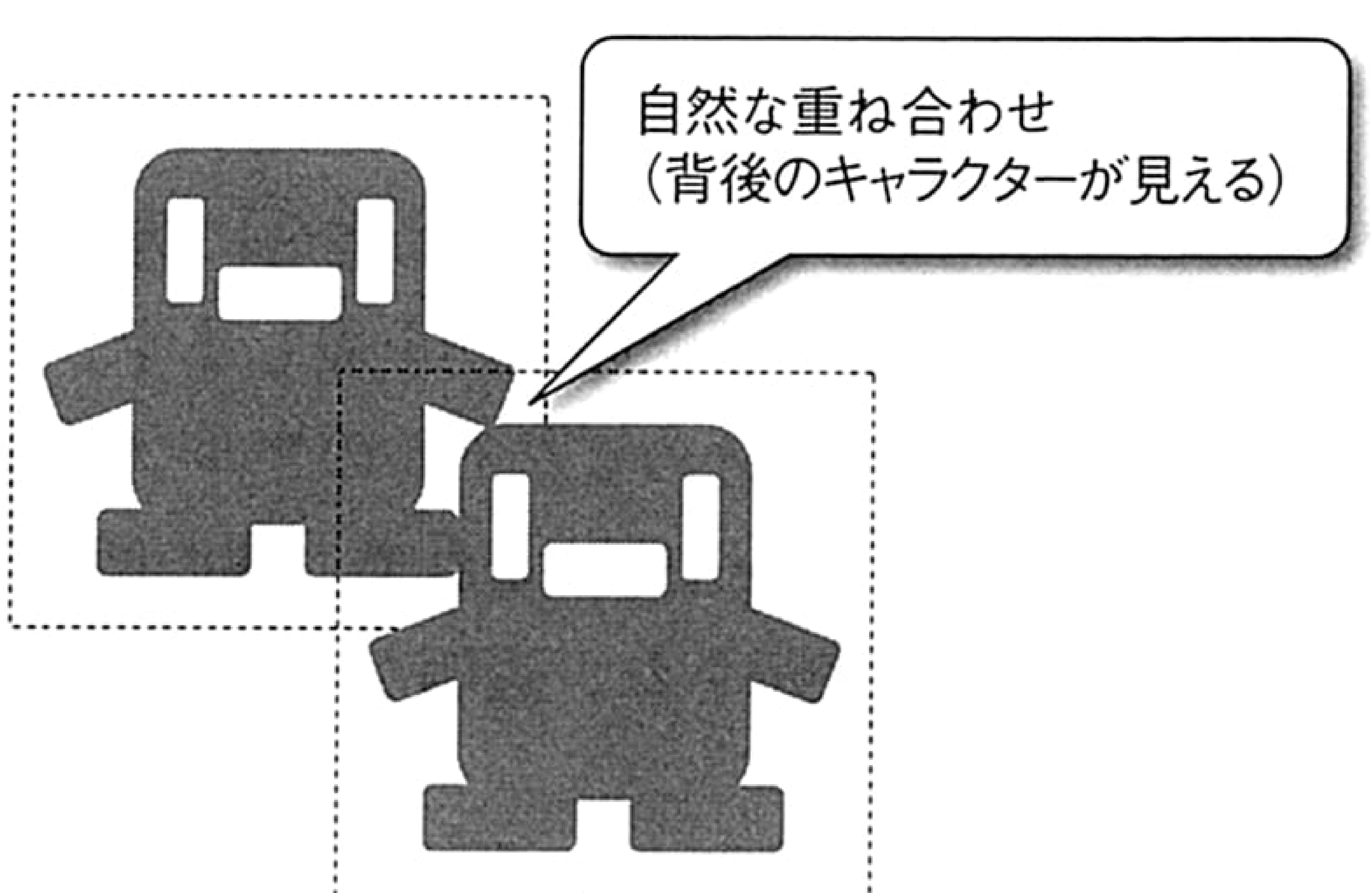


キャラクターを単体で表示したときには分からないのですが、他の画像を重ね合わせてみると、不自然な状態になってしまいます(Fig.11)。本来はキャラクターの形に沿って、重ね合わせた他の画像が見えるべきです(Fig.12)。

▼ **Fig.11**: 不自然な重ね合わせ

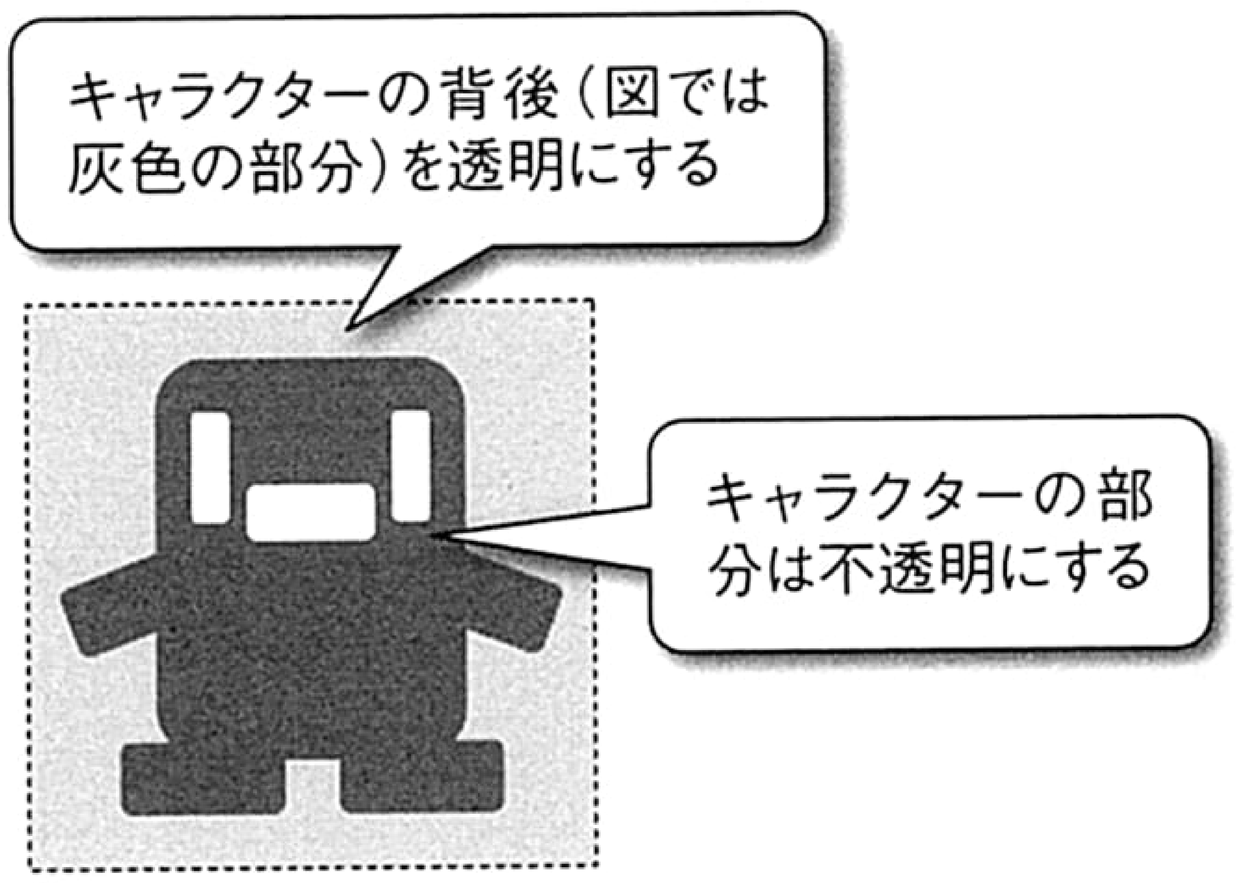


▼ **Fig.12**: 自然な重ね合わせ



キャラクターを自然に重ね合わせるためには、キャラクターの背後を透明にする必要があります(Fig.13)。画像全体としては矩形ですが、キャラクターの部分は不透明にして、キャラクター以外の部分は透明にします。

▼ **Fig.13**: 背後を透明にする

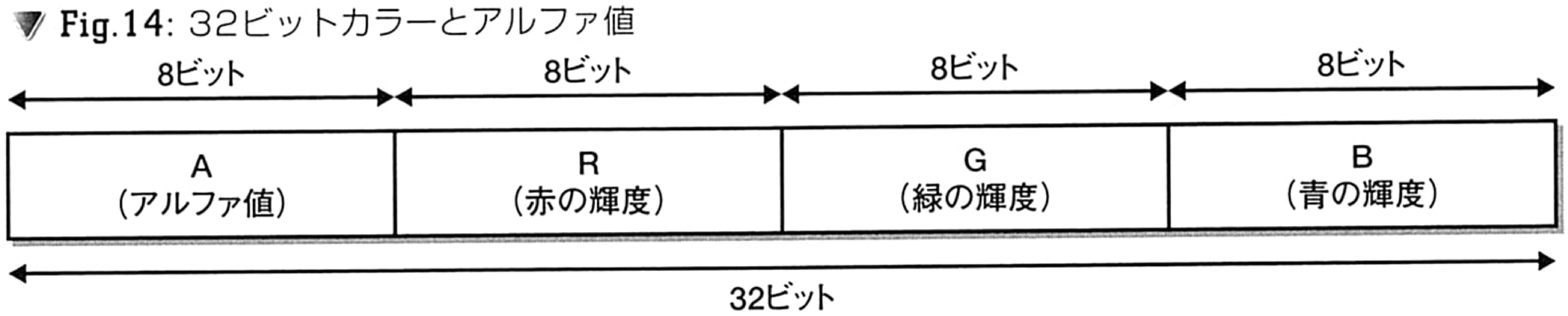


アルファ値

画像に透明な部分と不透明な部分を作るには、アルファ値という機能を使います。アルファ値は、ピクセル(画像を構成する点)の不透明度を表す値です。

一般的な32ビットフルカラー(約1677万色)の画像では、ピクセルのRGB値(光の三原色である赤・緑・青の輝度)を各8ビットで表現します。8ビットでRGBを表すと計24ビットなので、8ビットが残ります。この8ビットを使って、アルファ値を表現します(Fig.14)。

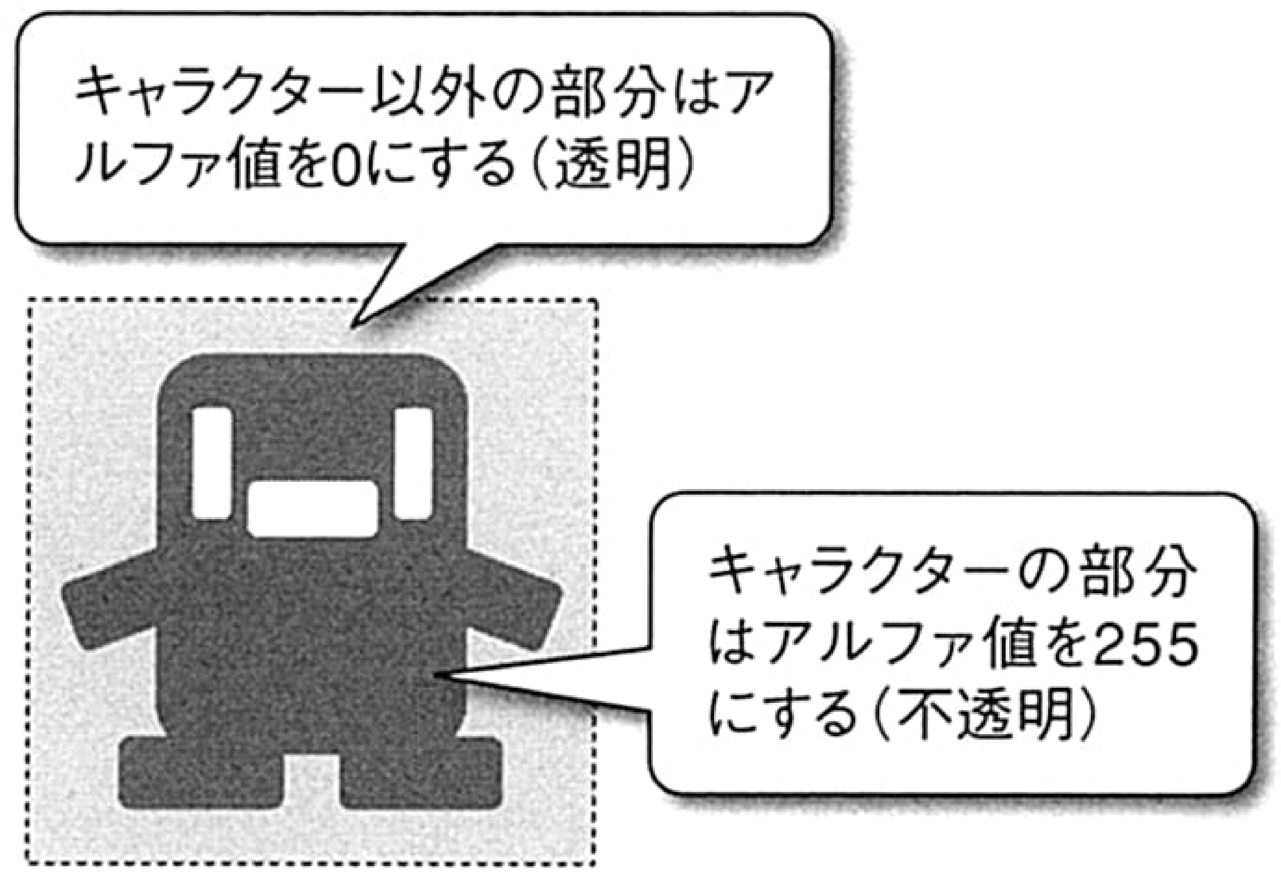
▼ **Fig.14**: 32ビットカラーとアルファ値



アルファ値を8ビットで表現すると、値の範囲は0から255までです。0は完全に透明な状態、255は完全に不透明な状態を表します。1〜254は、中間的な半透明の状態を表します。

キャラクターの背後を透明にする場合には、キャラクター部分のピクセルのアルファ値を255に、キャラクター以外のピクセルのアルファ値を0にします(Fig.15)。これで、他の画像と重ね合わせたときには、不透明なキャラクター部分はそのまま表示され、透明なキャラクター以外の部分には他の画像が表示されます。

▼ **Fig.15**: アルファ値を使って背後を透明にする

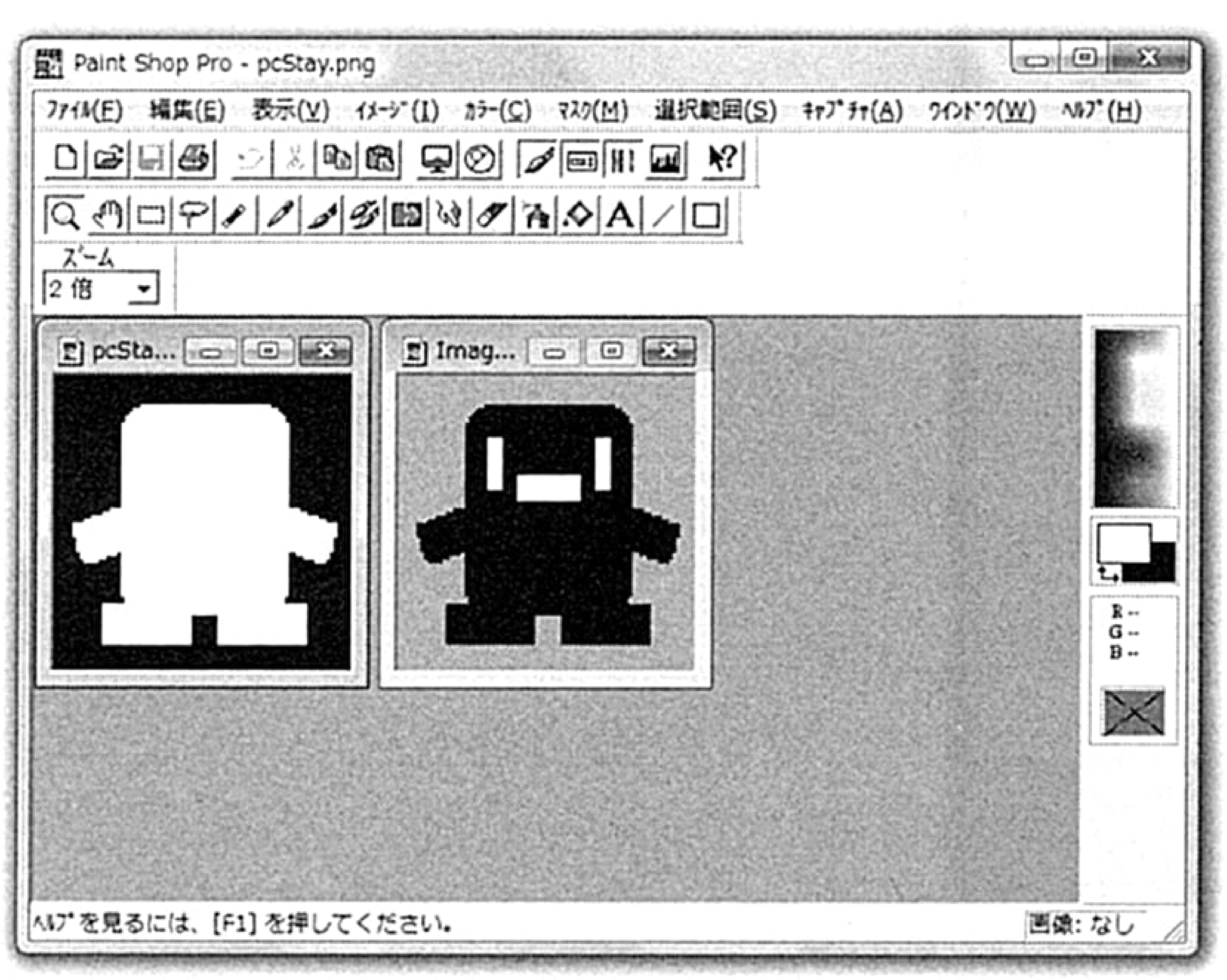


アルファ値を使った画像を保存するときには、アルファ値に対応したファイル形式を使う必要があります。ゲーム用の画像は、例えばPNG形式で保存するとよいでしょう。BMP（Microsoft Windows Bitmap Image）形式や、JPEG（Joint Photographic Experts Group）形式は、アルファ値を使う用途には適していません。

PNG形式の画像は、PNG形式に対応した画像ツールを使って作ります。Windows付属のペイントもPNG形式で画像を保存できますが、アルファ値の編集ができないため、他のツールを用意するとよいでしょう。

アルファ値の編集方法は、画像ツールによって異なります。本書で用いたPaint Shop Proの場合には、キャラクターの輪郭に合わせてマスクを作ります（Fig.16）。白い部分のアルファ値が255（不透明）、黒い部分のアルファ値が0（透明）になります。このマスクと一緒に画像をPNG形式で保存すると、背後が透明になったキャラクターの画像ができあがります。

▼ **Fig.16**: アルファ値の編集例

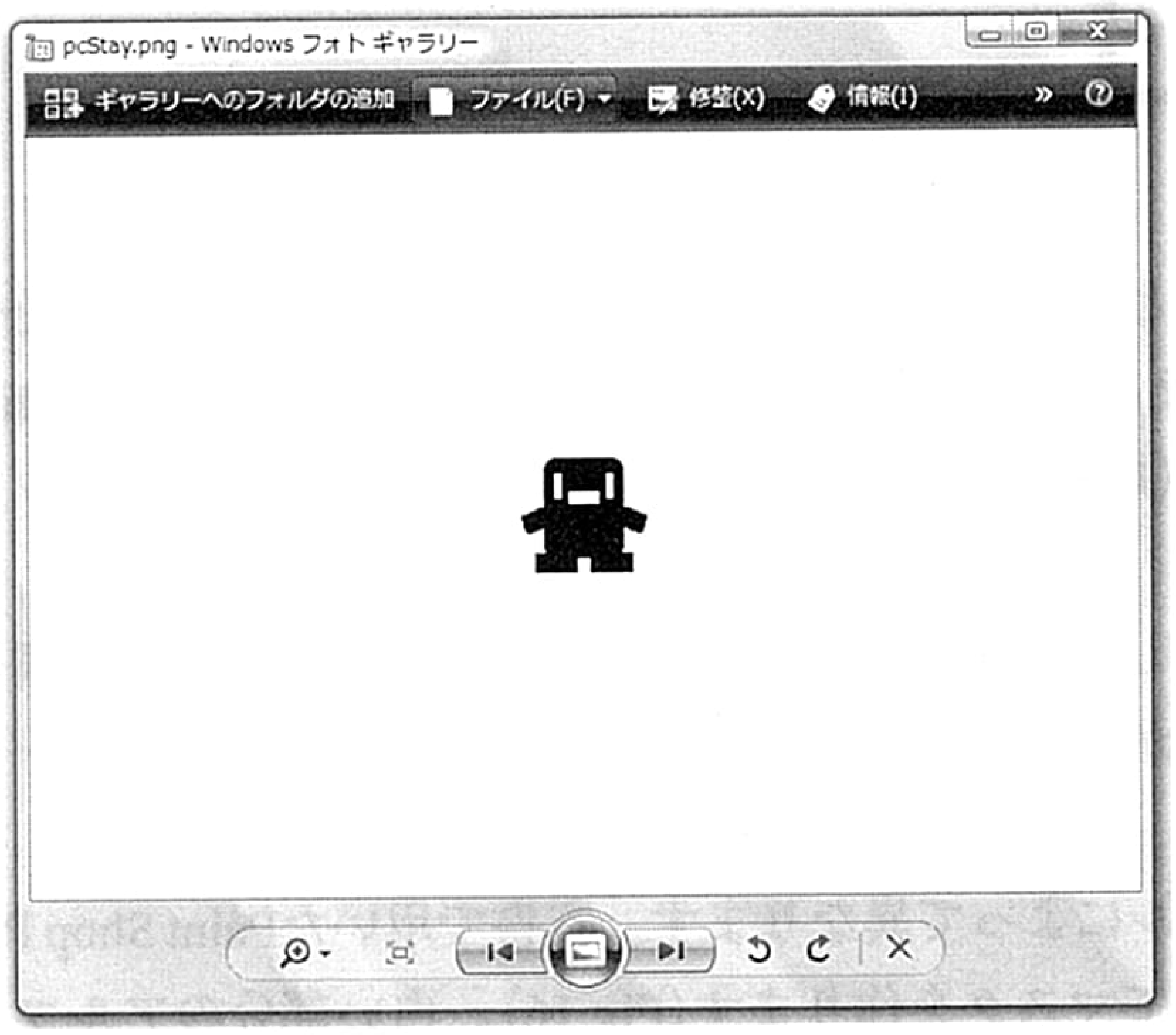


キャラクターの背後をきちんと透明にできたかどうかを確認するには、Windowsのエクスプローラから、画像をプレビューしてみるとよいでしょう。キャラクターの背後を白以外の色に

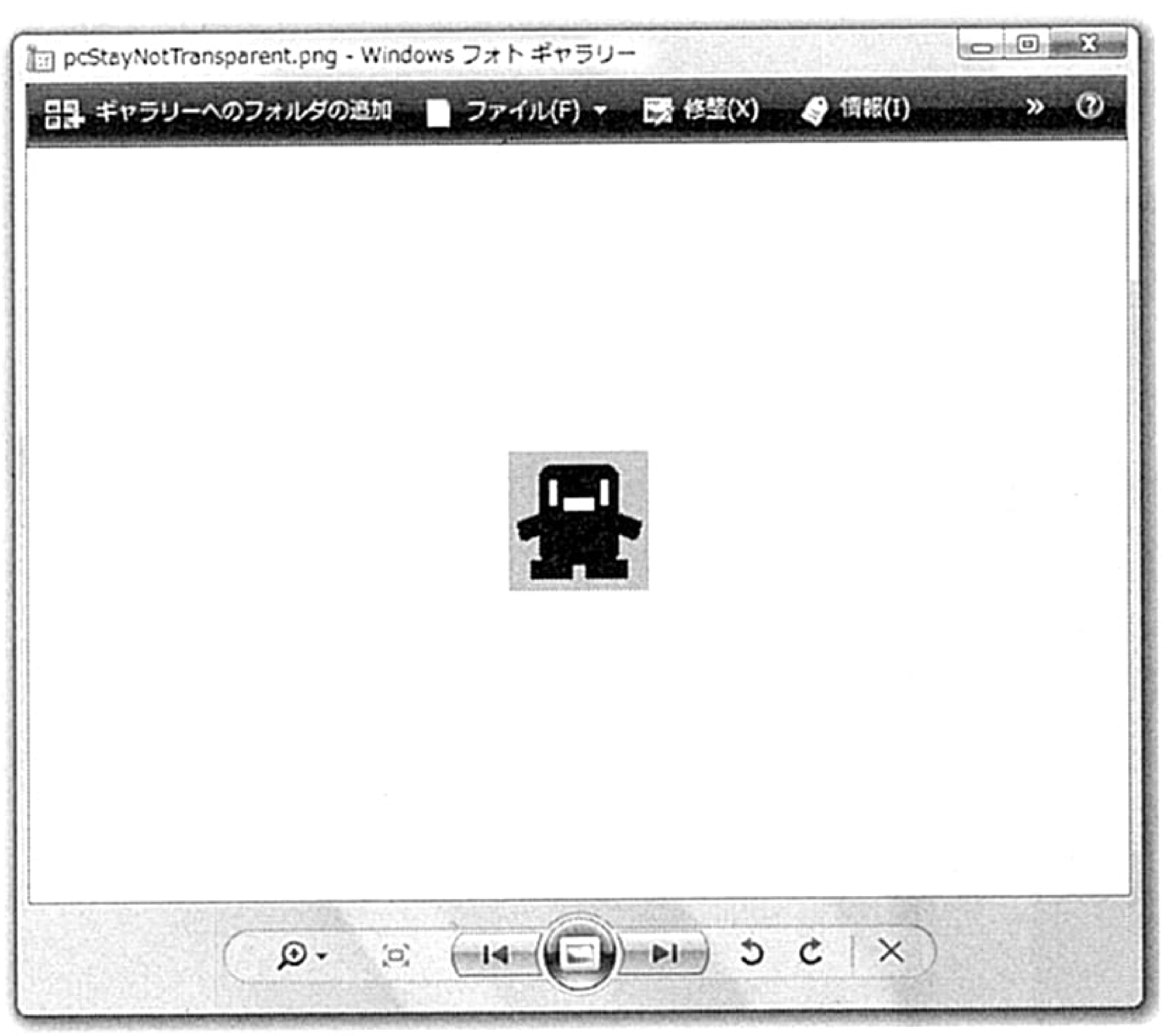
しておくと、透明になったかどうかが分かりやすくなります。

ここでは、キャラクターの背後を灰色にしておきます。画像をプレビューしたとき、背後が透明になっていれば、灰色の部分は表示されずに、背景の色が見えます(Fig.17)。背後が透明になっていないと、灰色の部分が表示されます(Fig.18)。正しく透明になっていないときには、画像ツールに戻って、背後が透明になるように修正します。

▼ **Fig.17**: 背後が透明のとき



▼ **Fig.18**: 背後が透明でないとき



PNG(ピング)

PNG(Portable Network Graphics)は画像ファイル形式の1つです。ネットワーク上を中心として、広く利用されています。
PNGが登場する以前は、ネットワーク上でGIF(Graphics Interchange Format)形式が普及していました。しかし、GIF形式が利用している圧縮アルゴリズムLZWの特許権を持つUNISYS社が、GIF形式の利用に対して特許使用料を求めたため、利用者が減りました。
PNGはGIFに代わるフリーの画像ファイル形式として登場しました。GIFと同等の機能を持ちつつも、より多くの色数を扱うことができ、アルファ値にも対応しています。
PNGは画像の圧縮に対応しています。そのため、無圧縮のBMPに比べると、画像ファイルのサイズをずっと小さくすることができます。
また、JPEGは圧縮に対応していますが、不可逆圧縮(元の画像を完全には復元できない)です。ゲームに使うようなハッキリとしたデザインの画像の場合には、ノイズやモアレが入ってしまうため向いていません。PNGは可逆圧縮(元の画像を完全に復元できる)なので、圧縮による画質の劣化はありません。
なお、BMPやJPEGも可逆圧縮に対応していますが、あまり用いられていません。可逆圧縮にはPNGを、写真などの不可逆圧縮にはJPEGを用いることが一般的です。少し以前は、WindowsではBMPを使うことが多かったのですが、壁紙などもPNGに対応したので、PNGを使う機会が増えました。

キャラクターの表示

キャラクターの画像ができたので、今度はキャラクターを表示してみましょう。キャラクターを表示する一番簡単な方法は、HTMLを使った方法です(List1)。

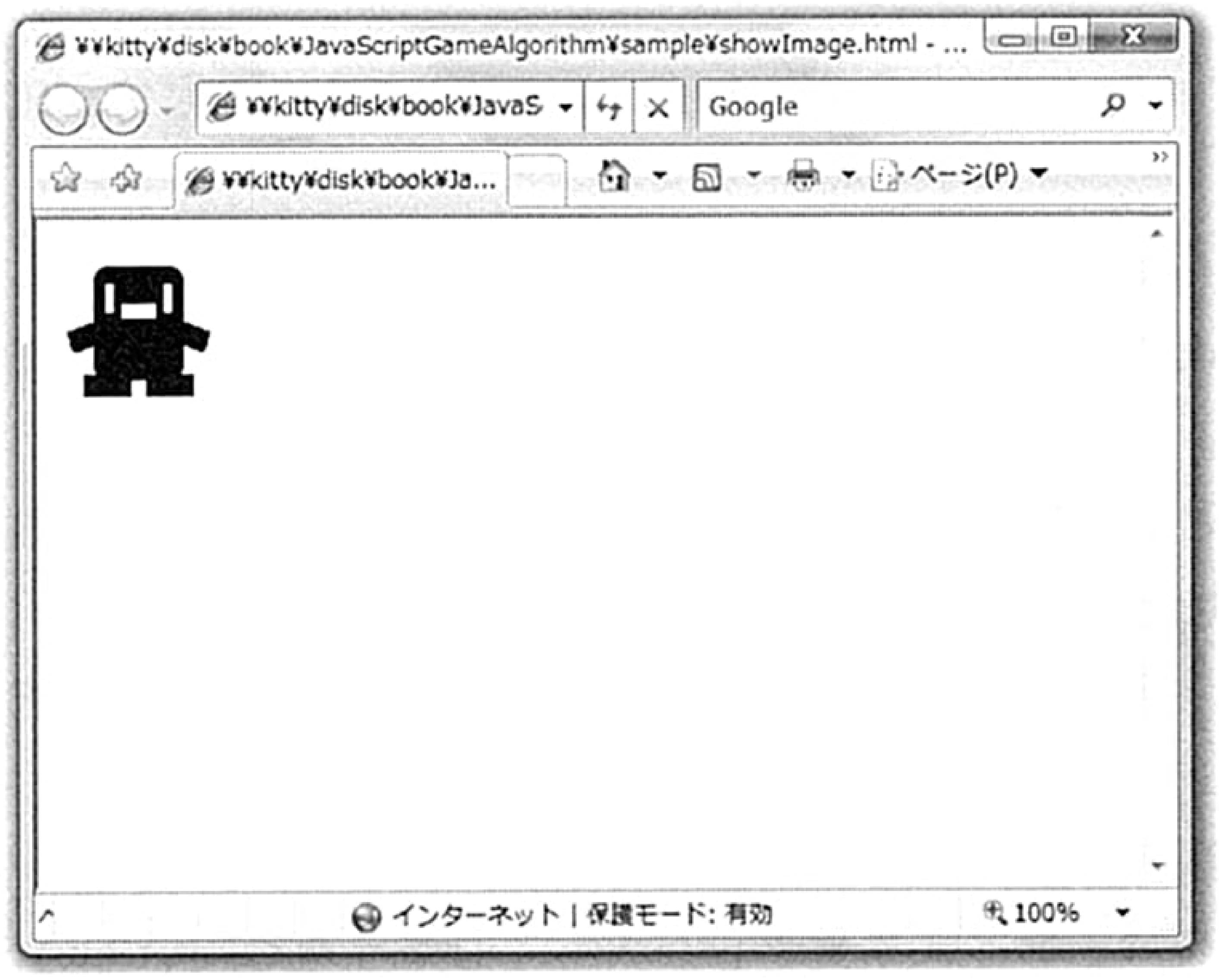
HTML(HyperText Markup Language)は、Webページを記述するための言語です。最近はブログの普及によって、HTMLを直接書く機会はやや減りましたが、決められたブログの形式とは異なるWebページを作りたいときには、HTMLを使うことがまだまだあります。

HTMLドキュメントを書くには、テキストエディタを使います。テキストエディタは市販品でも、フリーソフトウェアでも構いません。Windowsに付属するメモ帳でも大丈夫です。ただし、文字コードのUTF-8(Unicodeに関する符号化方式の一種)に対応しているテキストエディタを用意してください。メモ帳はUTF-8に対応しています。

HTMLドキュメントを書いたら、ファイルに保存します。ファイル名は、例えばshowImage.htmlのように、拡張子を.htmlとします。文字コードはシフトJISやEUCなどでも大丈夫ですが、本書ではUnicode(UTF-8)を使うことにしました。

HTMLファイルをブラウザで開くと、画像が表示されます(Fig.19)。ここではInternet Explorerを使いましたが、OperaやFirefoxといった他のブラウザでも構いません。

▼ **Fig.19**: ブラウザに表示された画像



HTMLでは<html>や<body>といったタグを使って、さまざまな要素を記述します。List1では<img>というタグを使って、キャラクターを表示しています。<img>は画像を表すタグで、src属性に画像ファイルを指定すると、その画像を表示することができます。

▼ **List1**: HTMLによるキャラクターの表示(showImage.html)

```
<html>
<body>
  <!-- 画像を表示する -->
  <img src="pcStay.png">
</body>
</html>
```

Unicode(ユニコード)

Unicodeとは、コンピュータ上で文字を表現するための文字コードの一種で、多言語の文字を単一の文字コードで扱えることが特徴です。最近のWindows、Linux、Mac OS XなどのOSは、いずれもUnicodeを標準の文字コードとして利用しています。

UTF-8

UTF-8はUnicodeの符号化方式(文字コードをバイト列で表現する方式)の一種です。Unicodeが定義する文字を、1バイトから6バイトまでの可変長のバイト列で表現します。Unicodeによるファイルへの入出力やネットワークでの送受信などでは、UTF-8が広く使われています。

JavaScriptを使う

HTMLを使うと、とても簡単にキャラクターを表示することができます。それでは、HTMLを使ってゲームを作ればよいかといえば、実はそうではありません。

もちろんHTMLでも、画像の表示やリンクなどを駆使すれば、色々なゲームを作ることができます。しかし、キャラクターが画面を動き回ったり、ボタン操作でジャンプしたりといった、リアルタイム性のあるゲームは苦手です。アドベンチャーやパズルといった動きが激しくないゲームは作れますが、アクションやシューティングといったゲームは、HTMLだけでは作ることが困難です。

そこで、本書ではJavaScriptを使います。JavaScriptはブラウザ上で動作するスクリプト言語で、現行のほとんどのブラウザ上で利用できます。実際に、検索サイトや通販サイト、ブログといったさまざまな種類のWebサイトで、JavaScriptが活用されています。

一方、同様に多くのWebサイトでは、コンテンツの制作にFlashを使っています。JavaScriptを使うか、Flashを使うかは迷うところですが、本書では次の理由からJavaScriptを採用しました。

・開発環境がフリー

Flashにもフリーの開発環境はありますが、Adobe社が提供する正式な開発環境は有償で、それなりに高価です。一方、JavaScriptはテキストエディタとブラウザがあれば開発できるので、お金をかけずに手軽に始めることができます。

・JavaScriptの知識をFlashに応用できる

FlashではActionScript（アクションスクリプト）というスクリプト言語を使いますが、このActionScriptとJavaScriptは、どちらもECMAScriptをベースにしています。そのため、JavaScriptでプログラミングを学んでおけば、その知識をFlashのプログラミングに活かすことができます。

JavaScript（ジャバスクリプト）

JavaScriptは、かつて広く使われていたブラウザのNetscape Navigator上で動作するスクリプト言語として開発されました。名前はプログラミング言語のJavaと似ていますが、営業上の理由でJavaという名前が付けられただけで、互換性は全くありません。
普及に伴い、JavaScriptの中核的な仕様はECMAScriptとして標準化されました。現在では、多くのブラウザでJavaScript（ECMAScript）を利用することができます。ただし、ブラウザによって非対応の機能があったり、動きが異なったりするので、プログラムを開発する際には各種のブラウザにおける動作テストが欠かせません。

ECMAScript（エクマスクリプト）

JavaScriptの中核的な仕様を標準化したスクリプト言語です。Netscape Navigator用に開発されたJavaScriptと、その方言としてInternet Explorer用に開発されたJScriptについて、両者の非互換性を解消するために作られました。
JavaScriptと、Flashのスクリプト言語であるActionScriptは、どちらもECMAScriptをベースにしています。ライブラリの機能は異なりますが、基本的なプログラミングの手法は共通です。

JavaScriptによるプログラミング

それでは、JavaScriptを使ってキャラクターを表示してみましょう。JavaScriptのプログラムは、HTMLドキュメントから呼び出して使います。その際に、

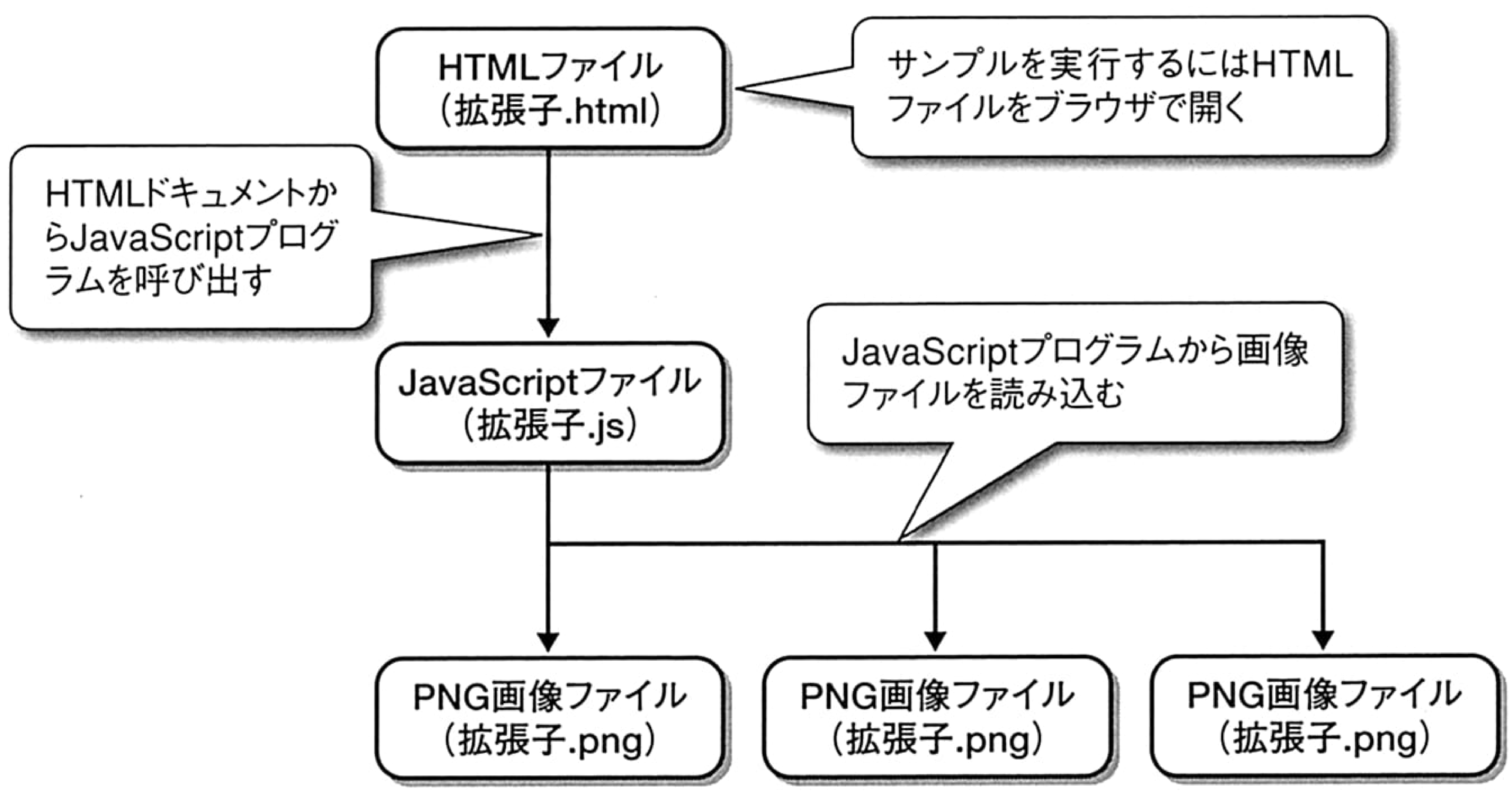
(1) HTMLドキュメントの中にJavaScriptのプログラムを書く
(2) HTMLドキュメントとは別のファイルにJavaScriptのプログラムを書く

という2つの方法がありますが、本書では(2)の方法を使うことにします。その理由は次の通りです。

- JavaScriptの長いプログラムは、HTMLファイルとは別のファイルにした方がすっきりして、扱いやすくなる
- いくつかのプログラムで共通する機能をファイルにまとめて、ライブラリ化することができる
- HTMLファイルを一度作ってしまえば、あとは色々なJavaScriptプログラムから、ほとんど変更せずに使い回すことができる

本書のサンプルは、1つのHTMLファイルと、いくつかのJavaScriptファイル、そして多くの画像ファイルから構成されています(Fig.20)。サンプルを実行するときには、HTMLファイルをブラウザで開きます。HTMLファイルはJavaScriptファイルを読み込んで、プログラムを実行します。画像ファイルは、JavaScriptのプログラムから読み込みます。

▼ **Fig.20**: サンプルのファイル構成



JavaScriptを呼び出すためのHTMLファイル

最初に、JavaScriptを呼び出すためのHTMLファイルを用意しましょう(List2)。HTMLファイルの作成にはテキストエディタを使います。ここではファイル名をshowImage2.html、文字コードはUTF-8としました。

先に作成したHTMLファイル(List1)はシンプルだったのですが、さまざまなブラウザでHTMLファイルを正しく読み込ませるためには、いくつかの決められた記述が必要です。これらの記述は少し煩雑に感じますが、一度書いてしまえば多くのHTMLファイルに流用できるので、それほど手間はかかりません。

ファイルの先頭には、

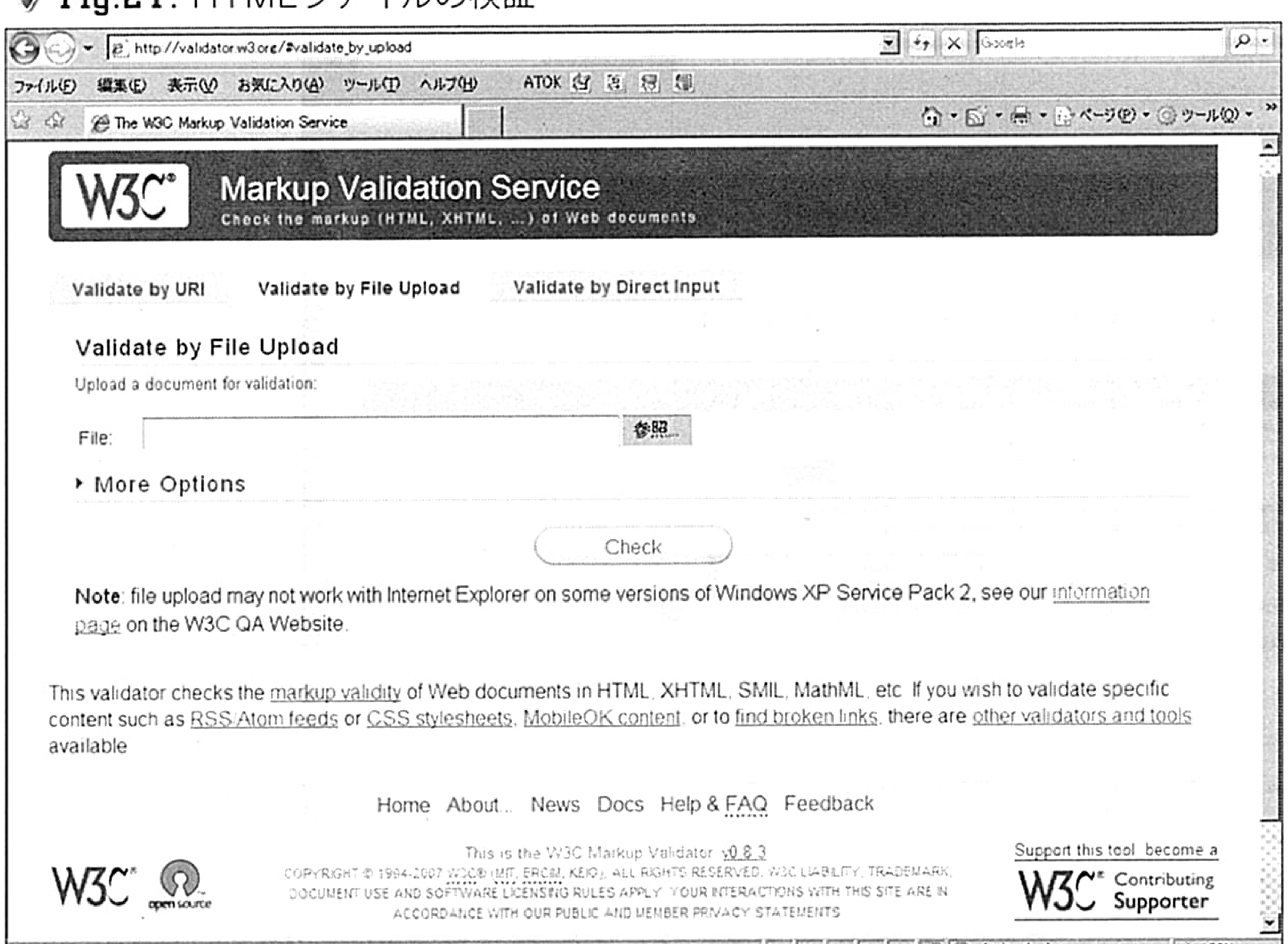
```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/
html4/loose.dtd">
```

のような文書型の宣言を書きます。ここでは、HTMLファイルがどのような仕様に基づいて書かれているのかを宣言します。本書ではHTML 4.01 Transitionalを採用しました。

なお、以下のWebサイト(The W3C Markup Validation Service)を利用すると、HTMLファイルが仕様に従っているかどうかを調べることができます(Fig.21)。

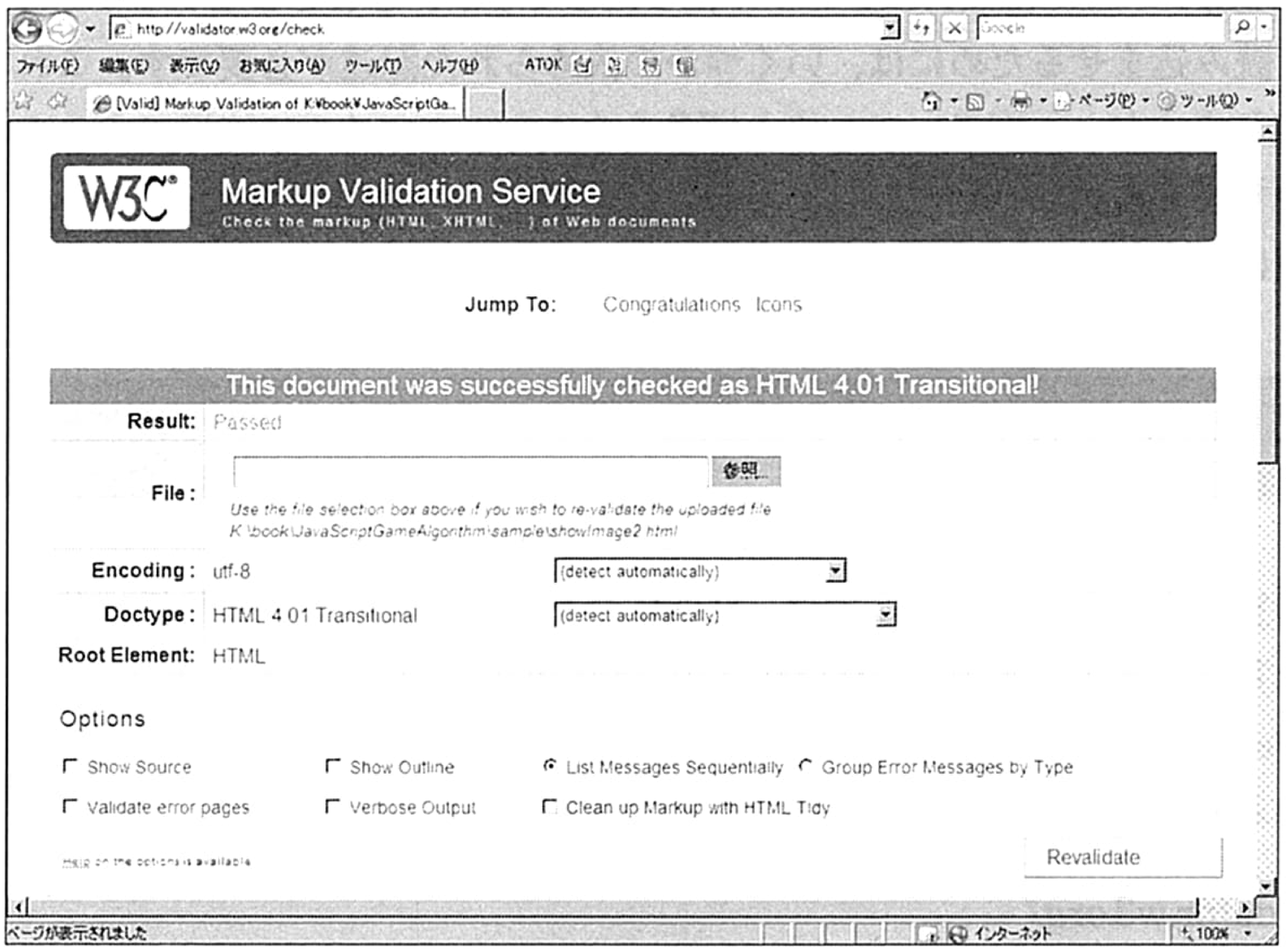
http://validator.w3.org/

▼ **Fig.21**: HTMLファイルの検証

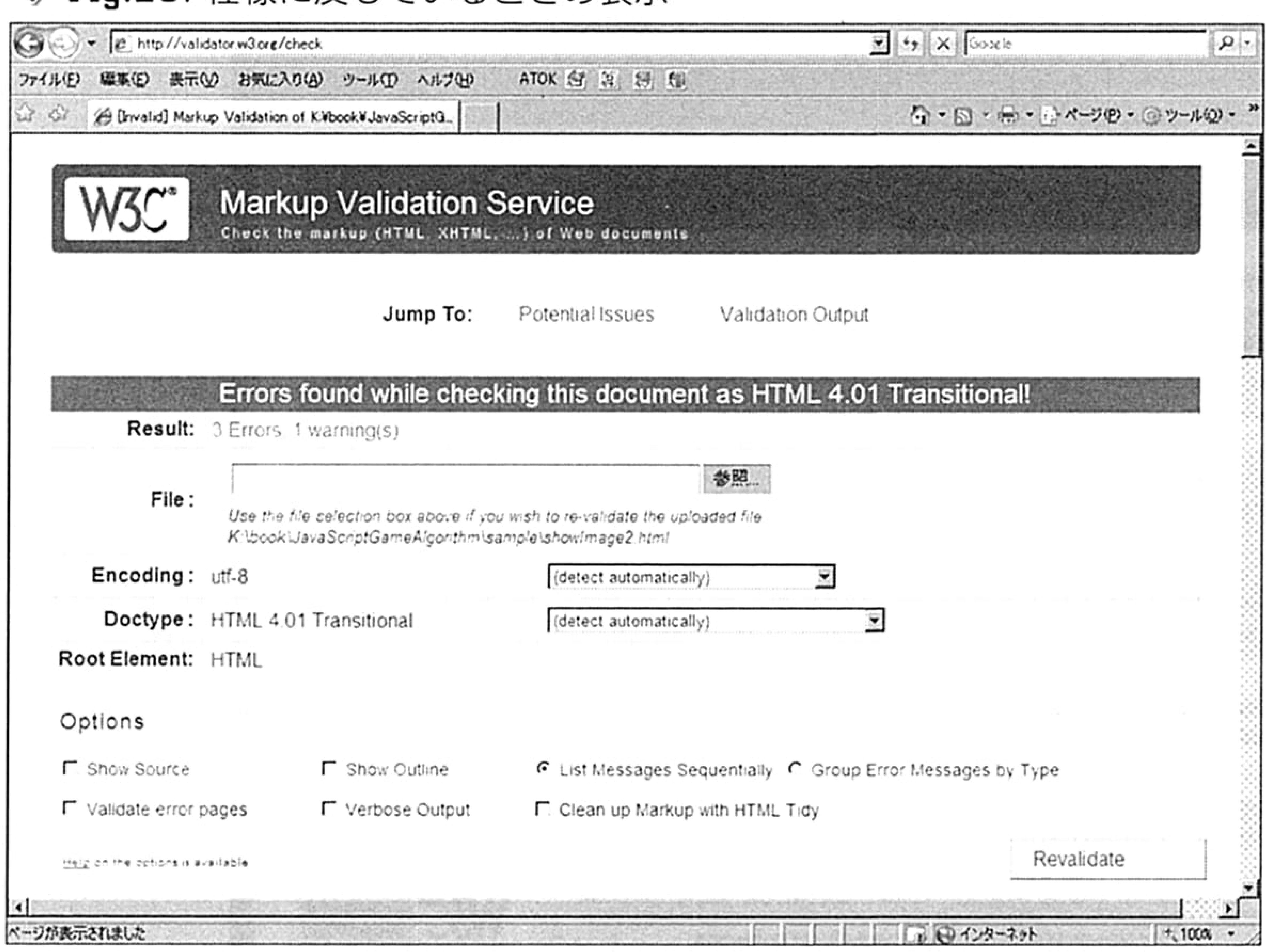


ファイルのURIを指定するか、アップロードするか、あるいはファイルの内容を直接入力して検証します。検証の結果、仕様に沿っていればその旨が表示され(Fig.22)、仕様に反している部分があればその内容が表示されます(Fig.23)。Operaのように、表示しているページに関して、この検証サービスを簡単に呼び出す機能を持ったブラウザもあります。

▼ **Fig.22**: 仕様に沿っているときの表示



▼ **Fig.23**: 仕様に反しているときの表示



次に、<html>タグを使ってHTML要素を記述します。HTML要素は、HTMLドキュメント全体を表します。HTML要素の内部では、<head>タグを使ってヘッド要素を、<body>タグを使ってボディ要素を記述します。

ヘッド要素では<meta>タグを使って、

```
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
```

のように書きます。ここでは、このHTMLファイルの文字コードがUTF-8であることを示しています。

続いて<script>タグを使って、

```
<script src="common.js" type="text/javascript" charset="UTF-8">
</script>
<script src="showImage2.js" type="text/javascript" charset="UTF-8">
</script>
```

のように、JavaScriptプログラムを読み込みます。ここではcommon.jsとshowImage2.jsという2つのJavaScriptファイルを読み込んでいます。

ヘッド要素の最後では、<title>タグを使って、

```
<title>キャラクターの表示</title>
```

のように、文書のタイトルを定義します。

次はボディ要素です。<body>タグでは、

```
<body onload="main()">
```

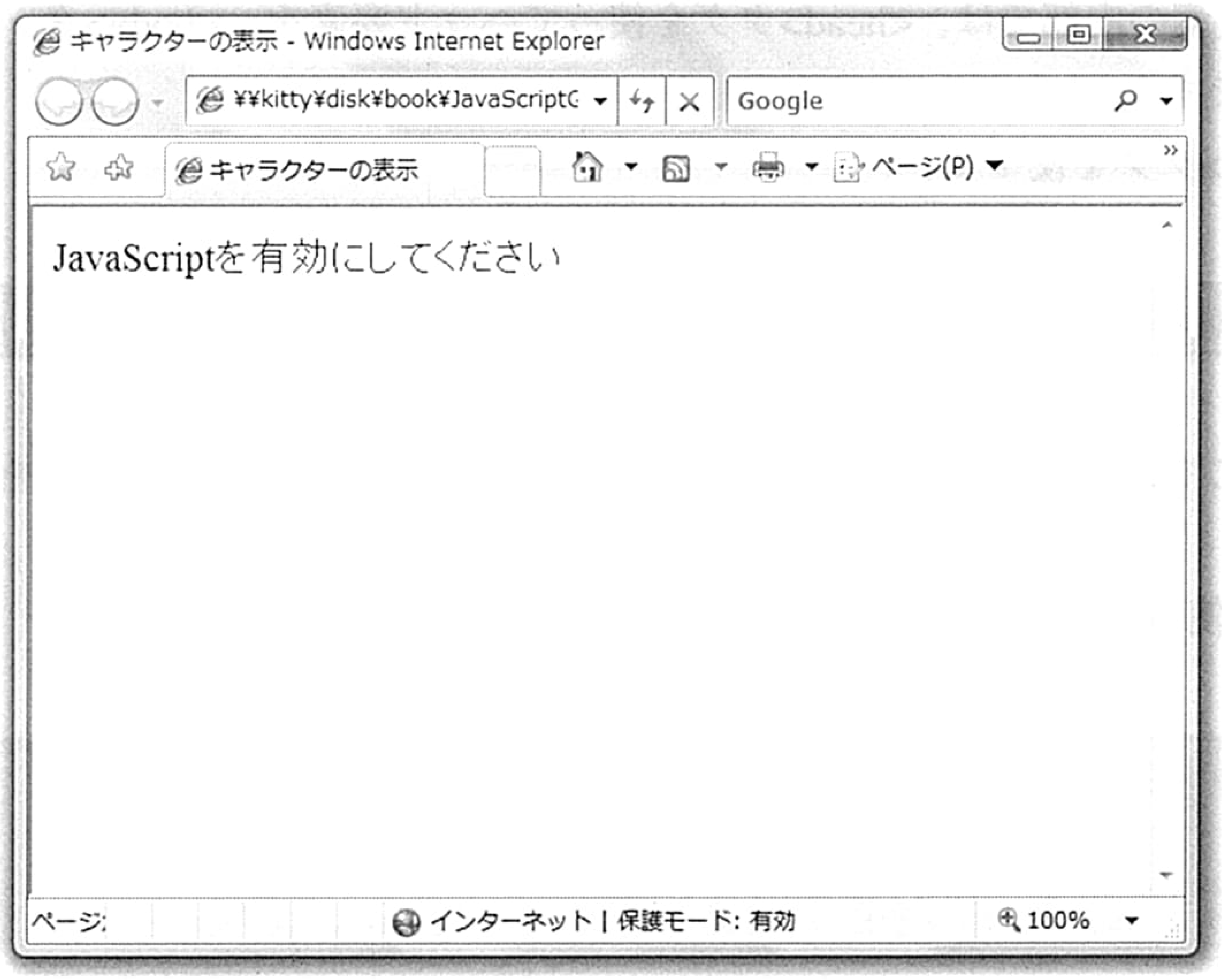
のように、onload属性を設定します。onload属性には、このHTMLファイルを読み込んだときに実行するJavaScriptのコマンドを指定します。この例のようにmain()と書くと、JavaScriptプログラムのmain関数を呼び出すことができます。関数については、後で詳しく説明します。

ボディ要素の内部では、<noscript>タグを使って、

```
<noscript>JavaScriptを有効にしてください</noscript>
```

のように記述しました。このメッセージは通常は表示されませんが、ブラウザでJavaScriptが無効になっていると表示されます。例えばInternet Explorerでセキュリティの設定を「高」にしておくと、メッセージが表示されます（Fig.24）。

▼ **Fig.24**: JavaScriptが無効になっているときのメッセージ



本書では多くのサンプルを紹介しますが、HTMLファイルの内容は、どのサンプルでもほとんど同じです。サンプルごとに異なるのは、<script>タグで読み込むJavaScriptファイル名だけです。他の部分は変わらないので、以後のサンプルについてはHTMLファイルは掲載せずに、JavaScriptファイルだけを掲載します。

▼ **List2**: JavaScriptを呼び出すためのHTMLファイル(showImage2.html)

```
<!-- 文書型の宣言 -->
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/
html4/loose.dtd">

<!-- HTML要素 -->
<html lang="ja">

  <!-- ヘッド要素 -->
  <head>

    <!-- 文字コード情報 -->
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">

    <!-- スクリプト (JavaScriptプログラム) の読み込み -->
    <script src="common.js" type="text/javascript" charset="UTF-8">
    </script>
    <script src="showImage2.js" type="text/javascript" charset="UTF-8">
    </script>
```

```
    <!-- 文書のタイトル -->
    <title>キャラクターの表示</title>
  </head>

  <!-- ボディ要素 -->
  <body onload="main()">

    <!-- JavaScriptが無効になっているときのメッセージ -->
    <noscript>JavaScriptを有効にしてください</noscript>
  </body>
</html>
```

HTML 4.01

HTML 4.01は現在主流のHTML仕様です。HTML 4.01には、Strict、Transitional、Framesetという3つのスキーマ(形式)があります。StrictはHTML 4.01本来の仕様です。TransitionalはHTML 3.2からの移行用で、HTML 4.01では非推奨となった機能も含んでいます。Framesetはフレームに関する機能を含めたものです。本書ではStrictではなく、Transitionalを採用しました。これはStrictに従った場合に比べて、Transitionalに従った方が、サンプルが簡潔になるためです。

URI(ユーアールアイ)

URI(Uniform Resource Identifier)は、一定の書式に基づいて色々なリソース(資源)を指し示すための仕組みです。URIが指し示すリソースはネットワーク上のファイルだけではなく、データベースやサーバなどを指し示すこともあります。

URL(ユーアールエル)

URL(Uniform Resource Locator)は、URIの一種で、インターネット上のリソースを指し示すための仕組みです。URLはWebサイトやWebページの場所を指す言葉として、広く普及しています。本書ではURIという言葉を使いますが、URLと読み替えても差し支えありません。

JavaScriptでキャラクターを表示する

次に、HTMLファイルから呼び出すJavaScriptプログラムを記述します(List3)。HTMLファイルと同様に、テキストエディタを使って書き、ファイルに保存します。JavaScriptファイルの拡張子は.jsです。ここではファイル名をshowImage2.jsとし、文字コードはUTF-8としました。

List3ではまず、

```
var img;
```

のように変数imgを定義します。変数はデータの記憶領域です。変数には、さまざまな値を代入（格納）することができます。

次に、main関数を定義します。関数とは、プログラムにおいて関連が深い処理をまとめたブロックです。JavaScriptでは、

```
function 関数名(引数) {
    処理;
}
```

のように、関数の定義を記述します。引数とは、関数に与えるパラメータ（数値や文字列などのデータ）です。

関数名は自由に決めて構いませんが、ここではList2（p.20）のボディ要素で、

```
<body onload="main()">
```

のように、JavaScriptのmain関数を呼び出すように記述したので、関数名をmainにします。

プログラムの主要な処理を行う部分のことを、メインルーチンと呼びます。mainという名前は、メインルーチンであることに由来しています。

main関数の内部では、

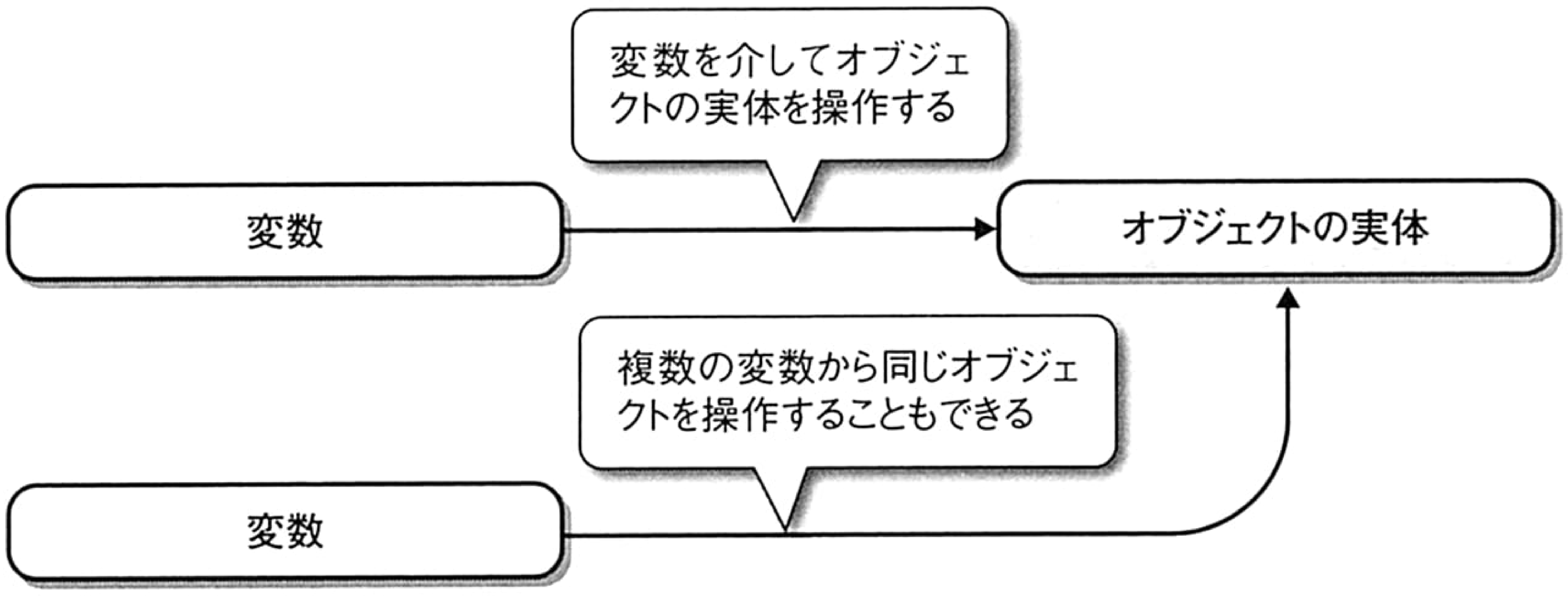
```
img=document.createElement("img");
```

のように、img要素を作成します。List1（p.14）で利用したように、img要素は画像を表示するための要素です。HTMLでは<img>タグを使ってimg要素を定義しますが、JavaScriptではプログラムからimg要素を作ることができます。

documentというのは、現在ブラウザで表示しているHTMLドキュメントを表すオブジェクトです。オブジェクトというのは、データと、そのデータに対する処理をまとめたものです。JavaScriptには、documentオブジェクトのほかにも、色々な機能を提供するオブジェクトが用意されています。

厳密な表現をすれば、documentというのはオブジェクト自体ではなく、documentという変数を介してオブジェクトを操作しているのに過ぎません（Fig.25）。オブジェクトの実体は変数とは別に存在しているので、例えば複数の変数から同じオブジェクトを操作するといったこともできます。ただし本書では便宜上、変数名を使って、documentオブジェクトのような呼び方をすることにします。

▼ **Fig.25**: 変数とオブジェクトの関係



createElementは、documentオブジェクトに属する関数です。HTMLのタグで定義するような、さまざまな要素を作るために使います。ここではdocumentオブジェクトのcreateElement関数を、"img"という引数で呼び出すことによって、img要素を作成しています。そして、作成した要素を変数imgに代入(格納)します。なおJavaScriptでは、"(ダブルクォーテーション)で囲んだ部分は文字列を表します。

続いて、

```
document.body.appendChild(img);
```

では、作成したimg要素を、HTMLドキュメントのボディ要素に追加します。ここで、

```
document.body
```

とは、documentオブジェクトに属するbodyオブジェクトを指します。これはHTMLドキュメントのボディ要素に相当するオブジェクトです。さらに、bodyオブジェクトに属するappendChild関数を呼び出すことによって、ボディ要素に新しい要素を追加します。

オブジェクトに属するオブジェクトや、オブジェクトに属する関数は、オブジェクトのプロパティ(またはメンバ)と呼びます。createElement関数やbodyオブジェクトはdocumentオブジェクトのプロパティで、appendChild関数はbodyオブジェクトのプロパティです。オブジェクトのプロパティを利用するには.(ピリオド)を使って、

```
オブジェクト.プロパティ
```

のように書きます。

main関数の最後では、

```
img.src="pcStay.png";
```

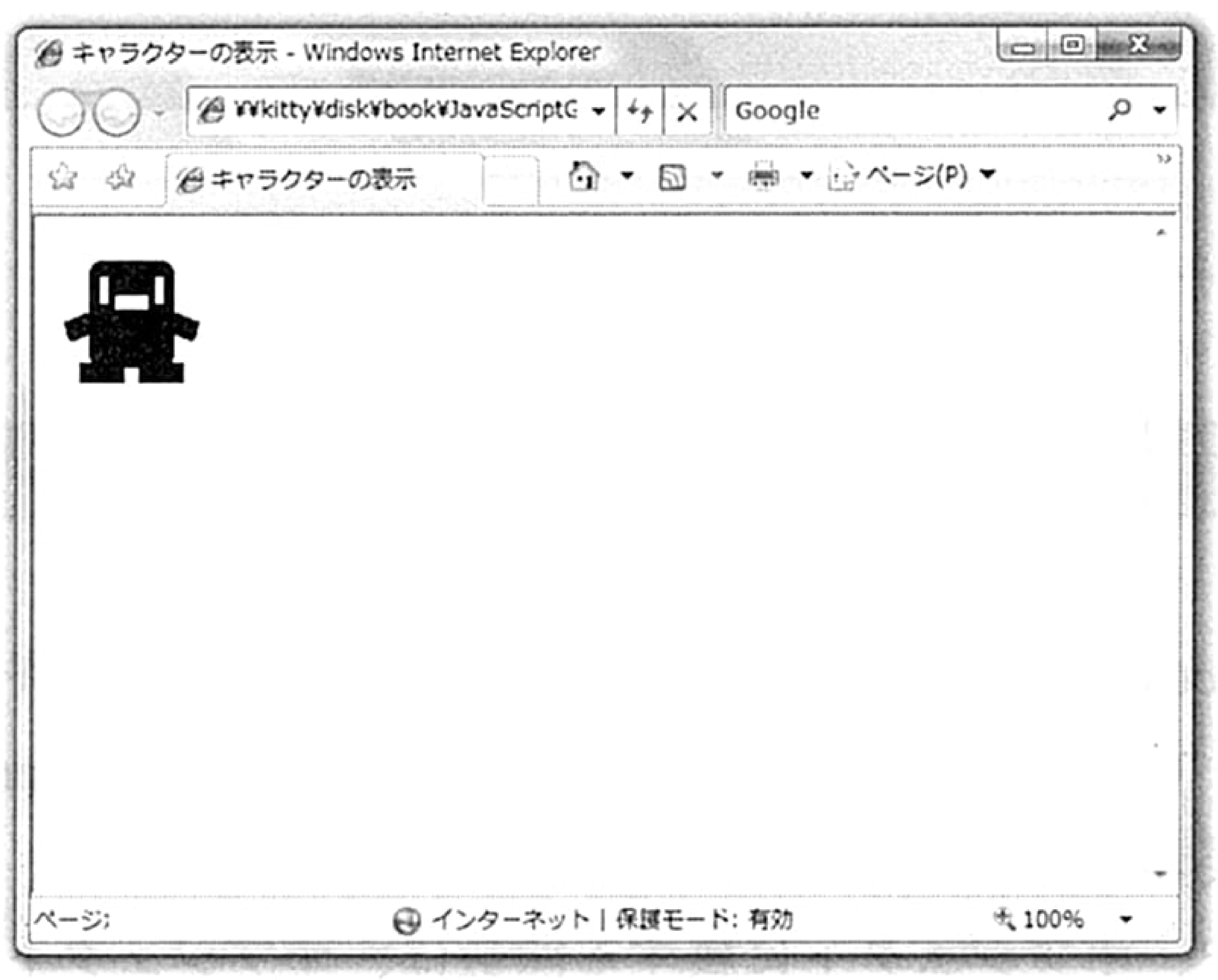
のように、imgオブジェクトのsrcプロパティに対して、"pcStay.png"という文字列を代入します。これは、img要素のsrc属性にpcStay.pngという画像ファイル名を設定することに相当します。

ここまでのプログラムで、

```
<body>
    <img src="pcStay.png">
</body>
```

のように、ボディ要素の内部にimg要素を作成し、画像ファイル名を指定したことになります。これで画像を表示することができます。List2（showImage2.html）をブラウザで開いてみてください。List1（showImage.html）と同様に、キャラクターが表示されるはずです（Fig.26）。

Fig.26: JavaScriptによるキャラクターの表示



なお、createElement関数やappendChild関数は、DOM（Document Object Model）と呼ばれる機能の一部です。DOMは、HTMLドキュメントやXMLドキュメントを、JavaScriptなどのプログラムから操作するための機能です。

List3: キャラクターを表示するJavaScriptプログラム（showImage2.js）

```
// 変数の定義
var img;

// main関数の定義
```

```
function main() {

  // img要素を生成する
  img=document.createElement("img");

  // img要素をbody要素に追加する
  document.body.appendChild(img);

  // img要素のsrc属性に画像ファイル名を設定する
  img.src="pcStay.png";
}
```

関数(かんすう)

関数とは、一連の処理手続きをまとめたブロックです。JavaScriptのほかにも、C/C++/Javaなどの多くのプログラミング言語が、関数をサポートしています。関数を使ってプログラムをいくつかのブロックに分けることにより、プログラムの構造を整理し、開発やメンテナンスを容易にすることができます。

変数(へんすう)

変数とは、データの記憶領域です。JavaScriptはもちろん、ほとんどのプログラミング言語が変数をサポートしています。変数に格納するデータを値(あたい)と呼び、変数に値を格納することを代入(だいにゅう)と呼びます。

型(かた)

プログラムが扱うデータには、整数・実数・文字・文字列といった種類があります。この種類のことをデータ型(がた)、または単に型と呼びます。変数にも値と同様に型があり、例えば整数型の変数には整数データを格納し、文字列型の変数には文字列データを格納するといった具合に、変数と値の型が対応しています。
C/C++/Javaなどのプログラミング言語では、変数と値の型を厳密に一致させる必要があります。変数の型とは異なる型の値を格納するときには、自動的に型の変換が行われる限られた場合を除き、値の型をプログラムで明示的に変換しなければなりません。
JavaScriptの場合、変数にはあらゆる型のデータを格納することができます。異なる型のデータを結合したり、異なる型のデータ間で演算を行うときには、自動的に型の変換が行われます。これはプログラムが組みやすい面もありますが、プログラムが思わぬ動きをして、見つけにくいバグになってしまう面もあります。

オブジェクト

オブジェクトとは、プログラムが扱うデータと、そのデータに対する操作をまとめた概念です。関数を使うと、プログラムの処理をブロックに分けて整理することができます。一方、オブジェクトを使うと、さらにデータを処理と一体化させた上で、分類して整理することができます。
オブジェクトを使ったプログラミング手法を、オブジェクト指向(しこう)プログラミングと呼びます。オブジェクト指向プログラミングに基づいたプログラミング言語は、オブジェクト指向プログラミング言語と呼びます。JavaScriptやC++/Javaなどは、オブジェクト指向プログラミング言語です。

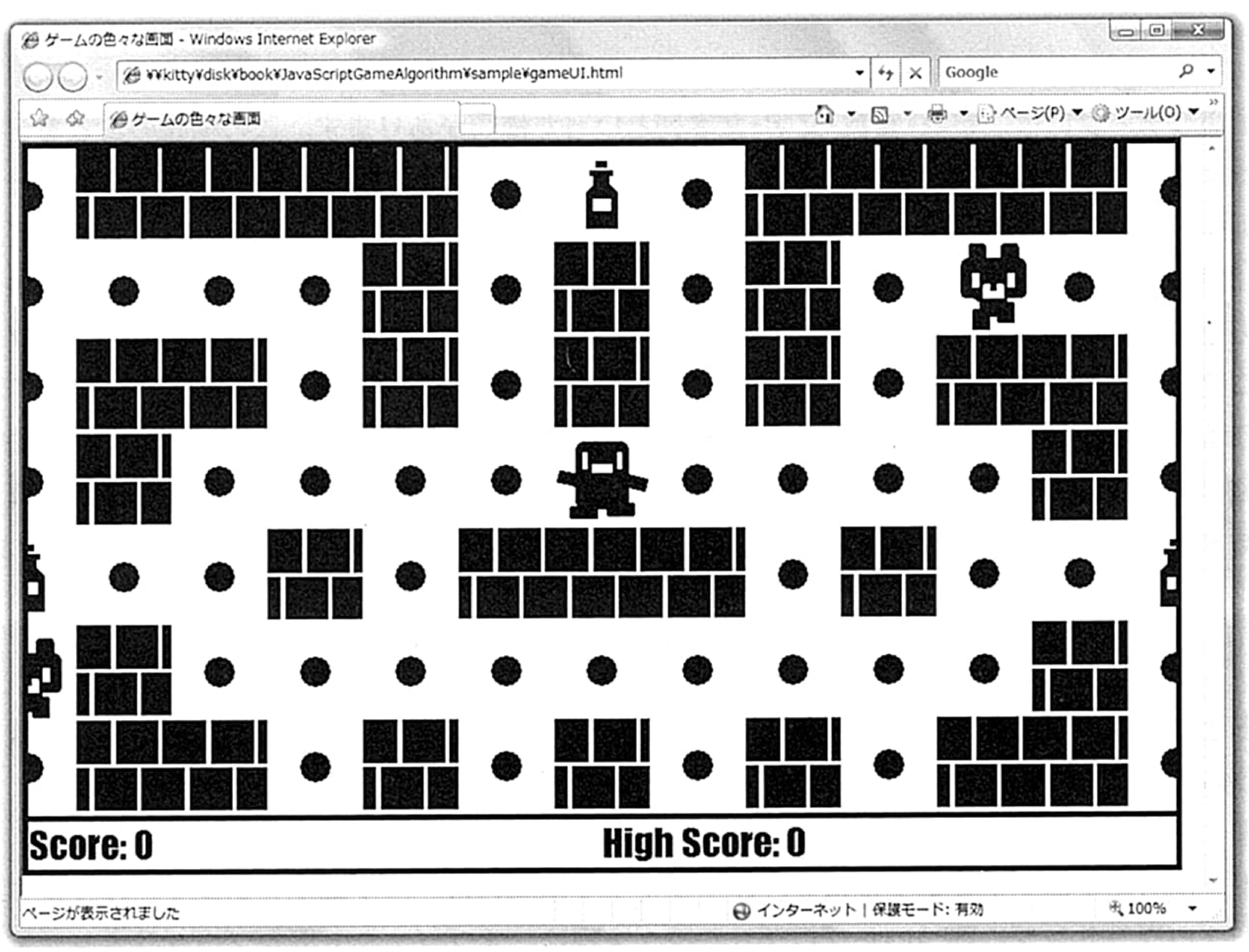
さまざまなブラウザへの対応

JavaScriptはほとんどのブラウザがサポートしていますが、実はブラウザによって非対応の機能があったり、ブラウザによって微妙に動きが違ったりします。プログラムがどのブラウザでも問題なく動くように作るには、全てのブラウザが対応している機能を選んで使うことと、こまめに各ブラウザで動作確認をすることが欠かせません。

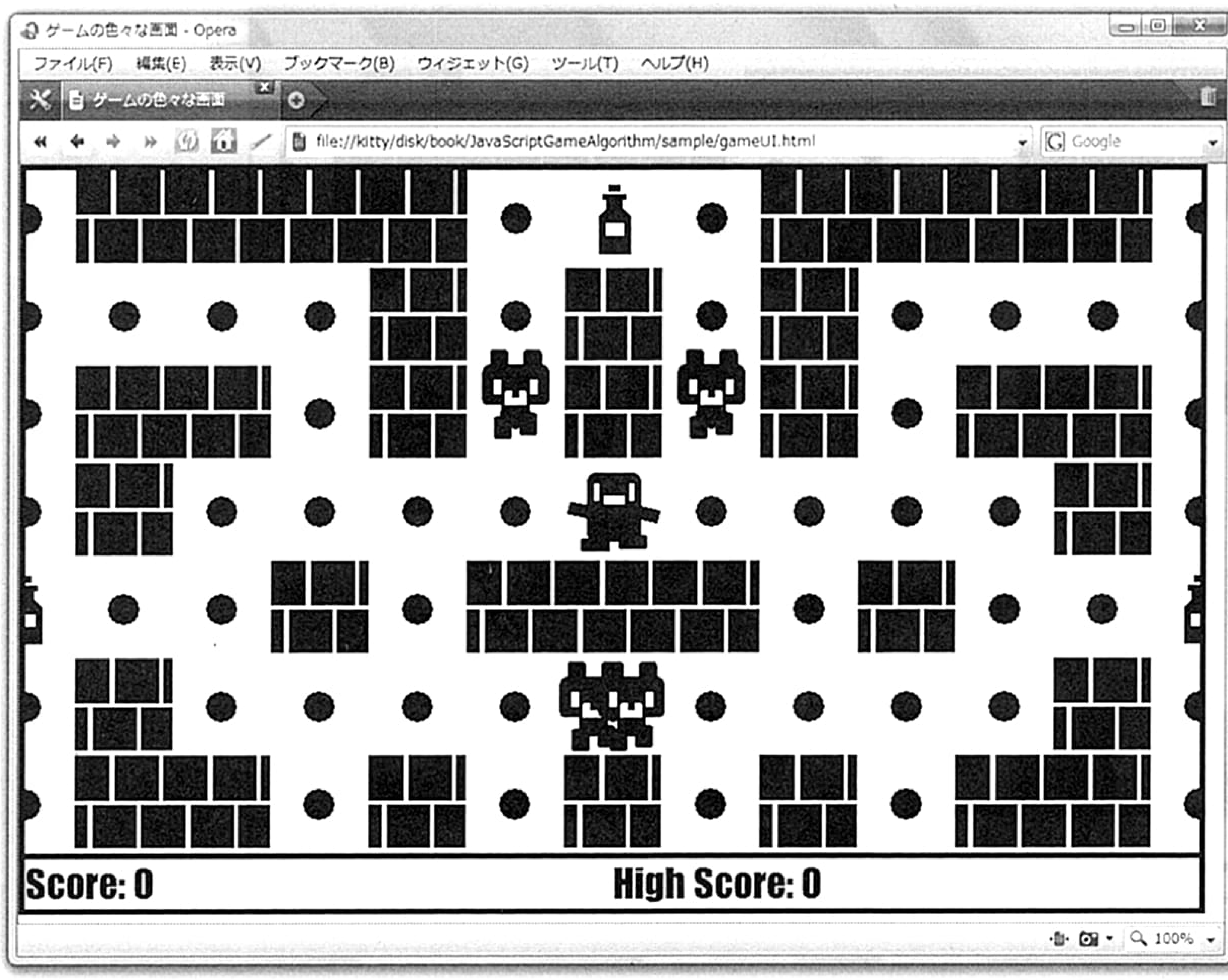
本書のサンプルは、できるだけ多くのブラウザで動作するように作成しました。対応ブラウザは以下の通りです。図は、同じサンプル(gameUI.html)を各ブラウザで実行したところです。動作確認はWindows Vista SP1とWindows XP SP3上で行ないました。

- Internet Explorer 7.0(Fig.27)
- Opera 9.6(Fig.28)
- Firefox 3.0(Fig.29)
- Safari 3.1(Fig.30)
- Google Chrome Beta 0.2(Fig.31)

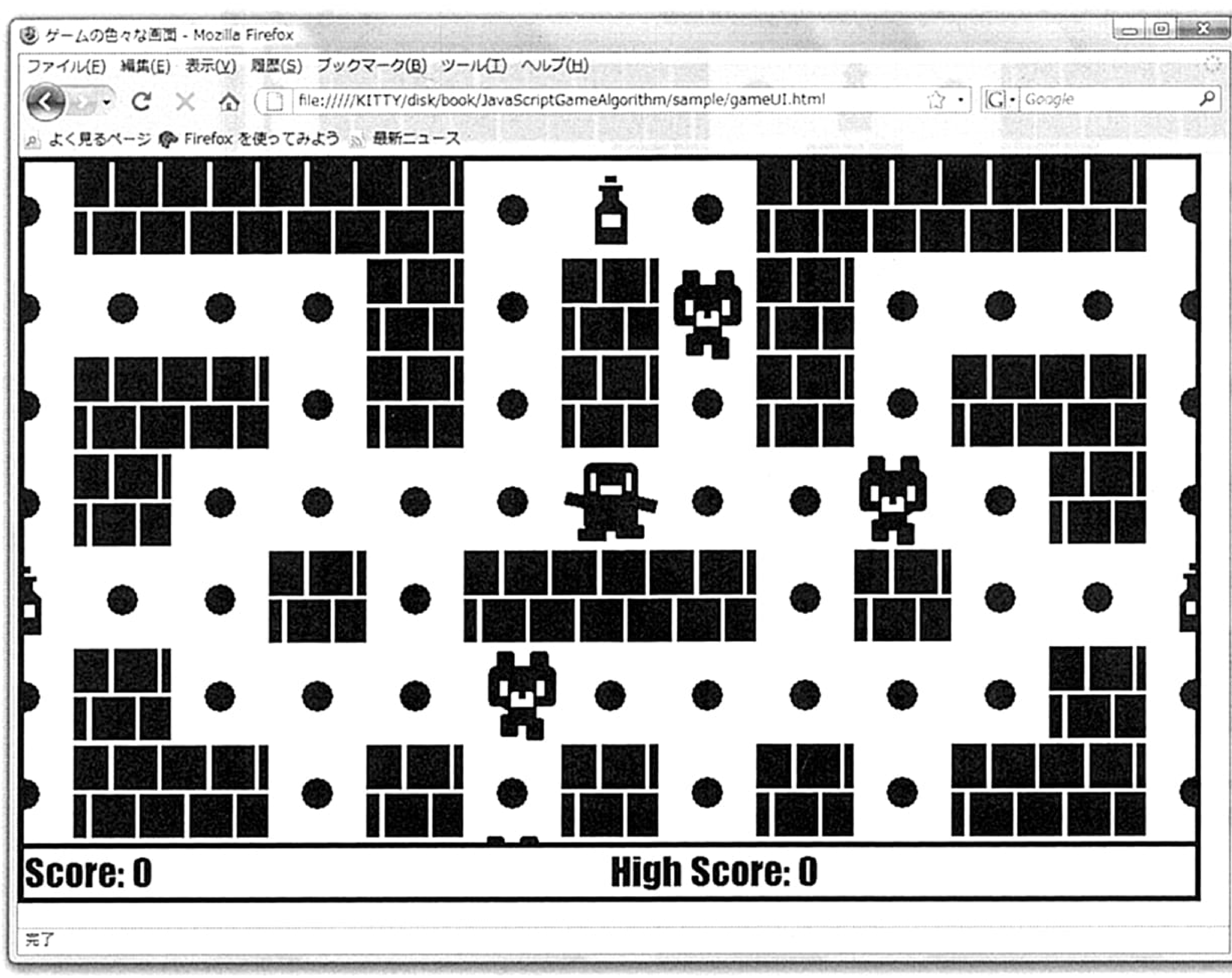
▼ Fig.27: Internet Explorerによる実行例



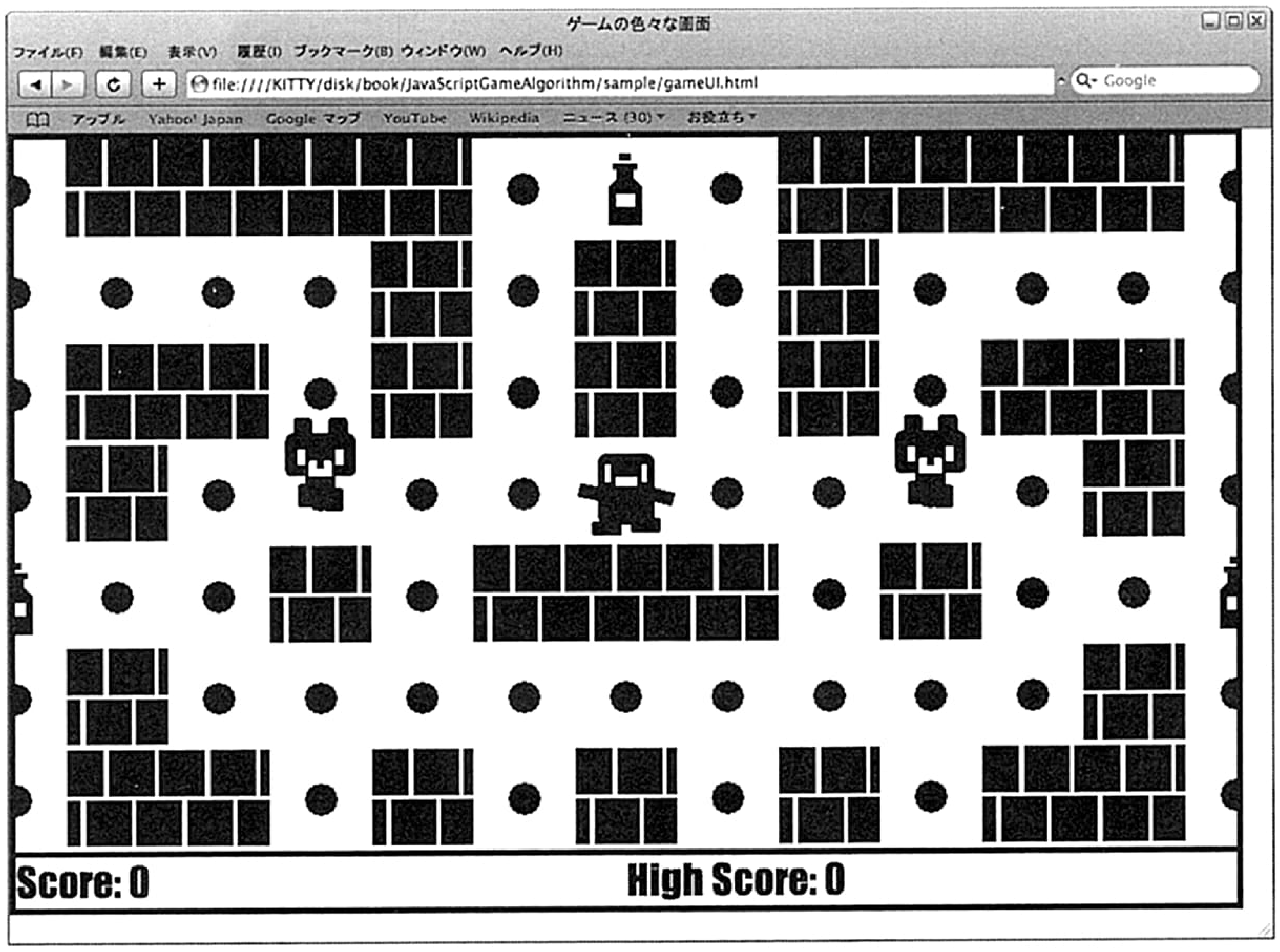
▼ **Fig.28**: Operaによる実行例



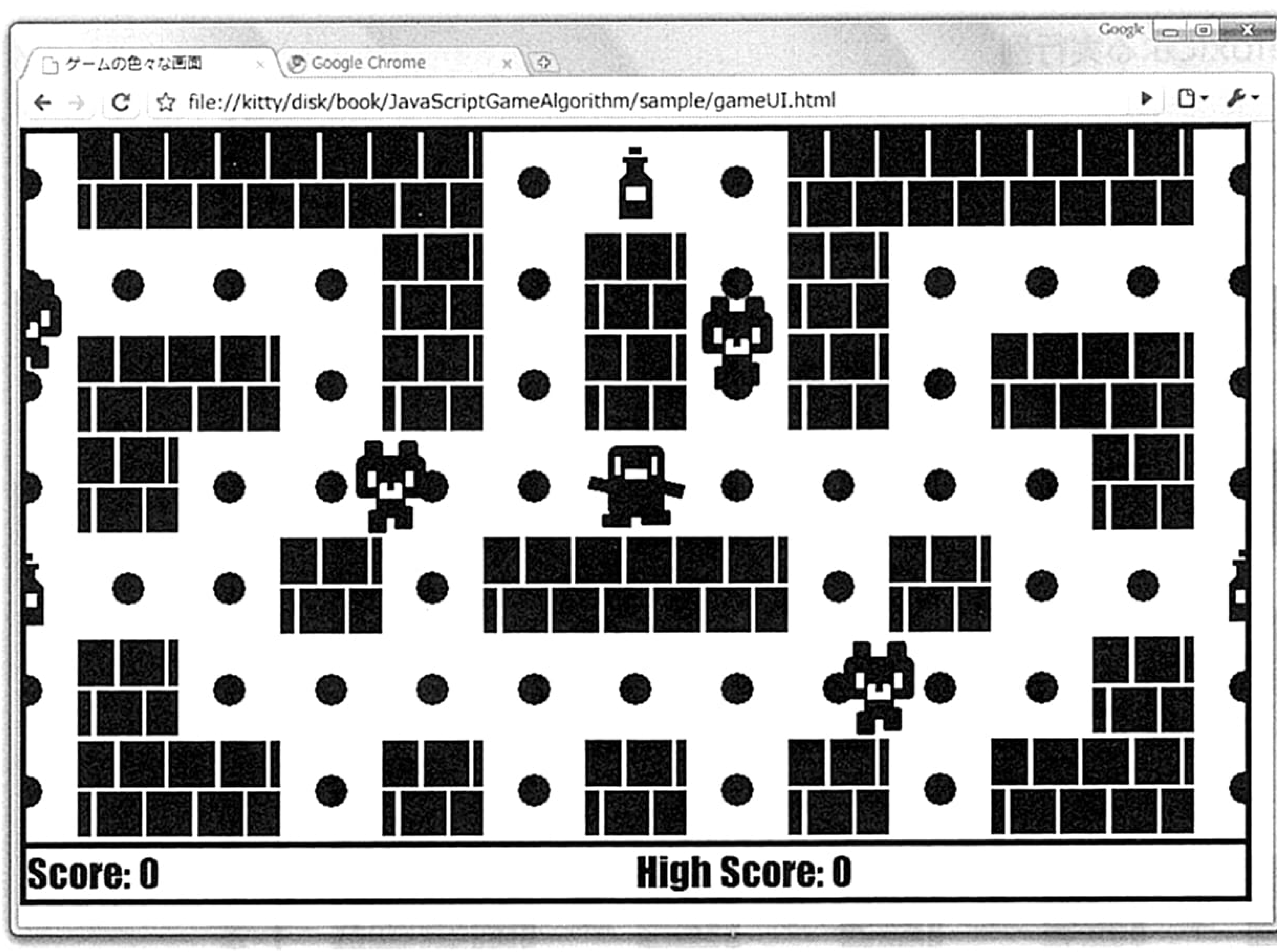
▼ **Fig.29**: Firefoxによる実行例



▼ **Fig.30**: Safariによる実行例



▼ **Fig.31**: Chromeによる実行例

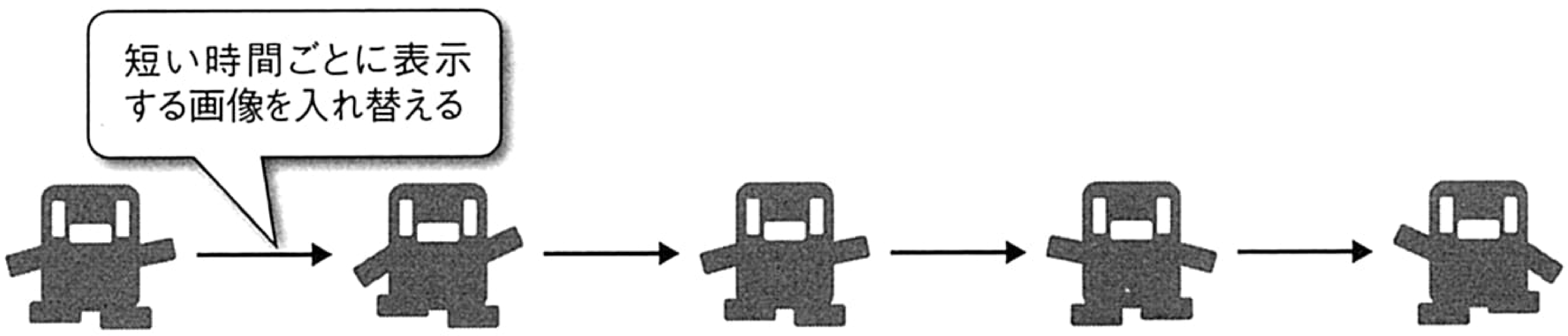


キャラクターのアニメーション

キャラクターを表示できたところで、今度はキャラクターをアニメーションさせてみましょう。キャラクターの位置は変えずに、その場で歩くアニメーションをさせることにします。

ゲームでキャラクターをアニメーションさせる方法は、映画やテレビのアニメーションと同じ仕組みです。映画やテレビでは、1/24秒や1/12秒といった短い時間ごとに、素早く絵を入れ替えることによって、キャラクターが動いているように見せます。ゲームの場合も全く同じで、短い時間ごとに表示する画像を入れ替えれば、アニメーションが実現できます(Fig.32)。

▼ **Fig.32**: アニメーションの仕組み

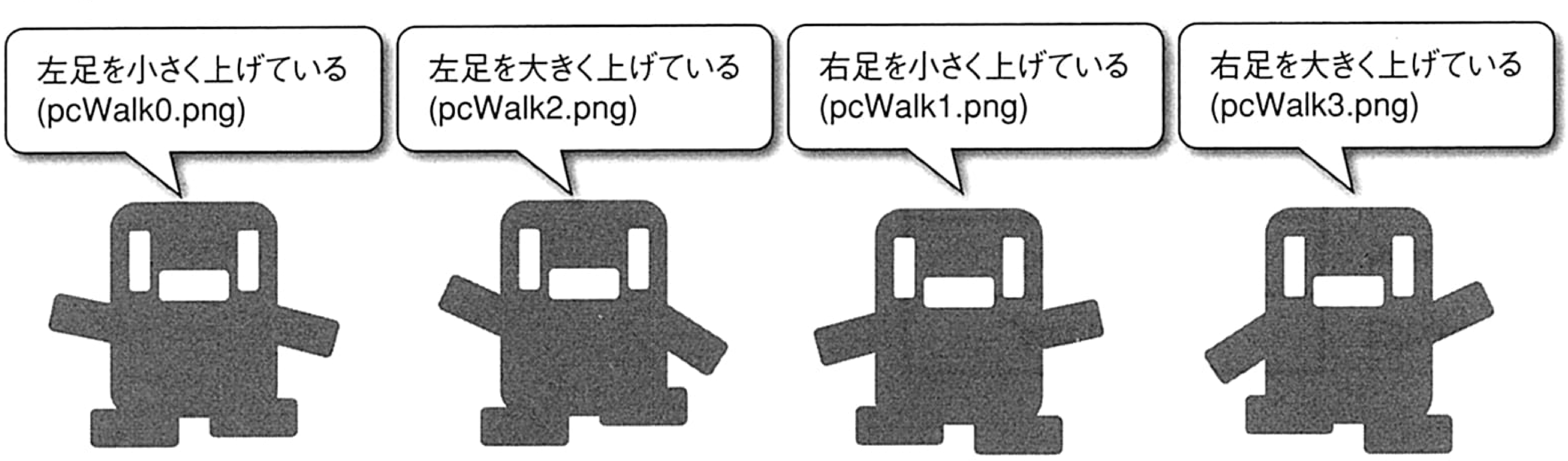


アニメーション用の画像

まず、アニメーション用のキャラクター画像を用意しましょう。最初に作ったキャラクターの画像と、作り方は全く同じです。ドローツールやペイントツールを使って、キャラクターの動きを表す何枚かの画像を描きます。

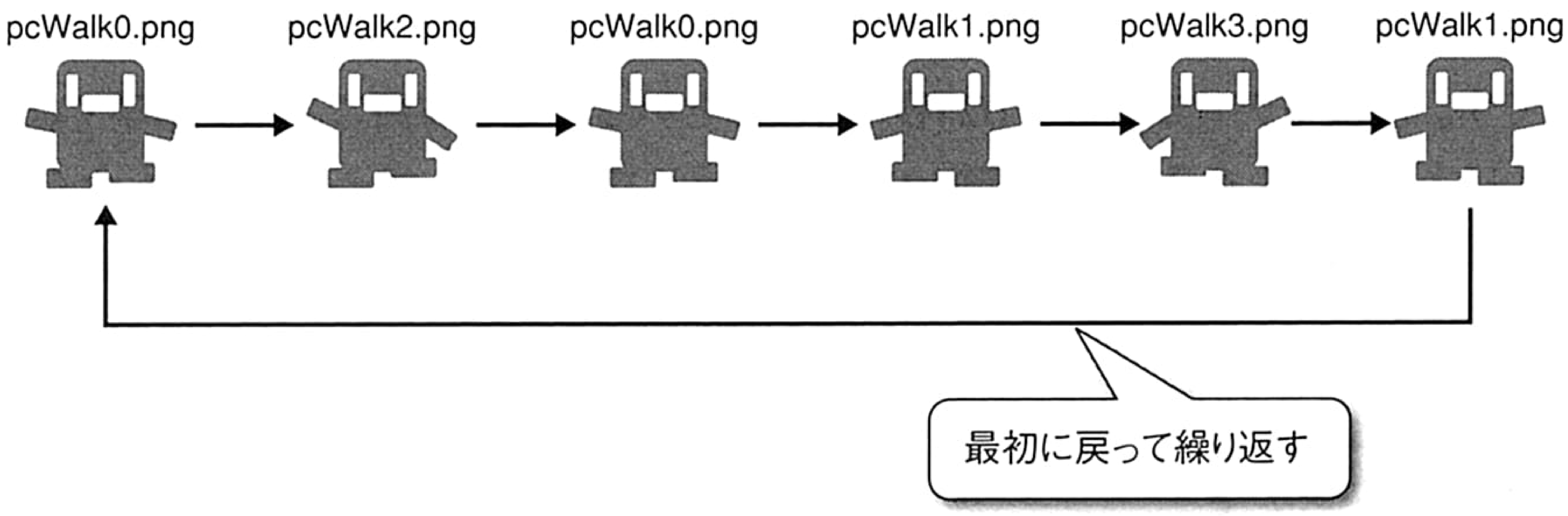
ここでは4枚の画像を用意しました(Fig.33)。キャラクターが手足を左右に上げている様子です。手足の上げ方が小さいものと大きいものを、左右それぞれに用意したので、合計4枚の画像になりました。ファイル名はpcWalk0.png～pcWalk3.pngとします。

▼ **Fig.33**: キャラクターが歩く画像



この画像を特定の順番で表示すると、キャラクターが歩いているアニメーションになります(Fig.34)。好みに応じてさらに多くの画像を用意すれば、より滑らかなアニメーションを作ることもできます。

▼ **Fig.34**: 歩くアニメーションの構成

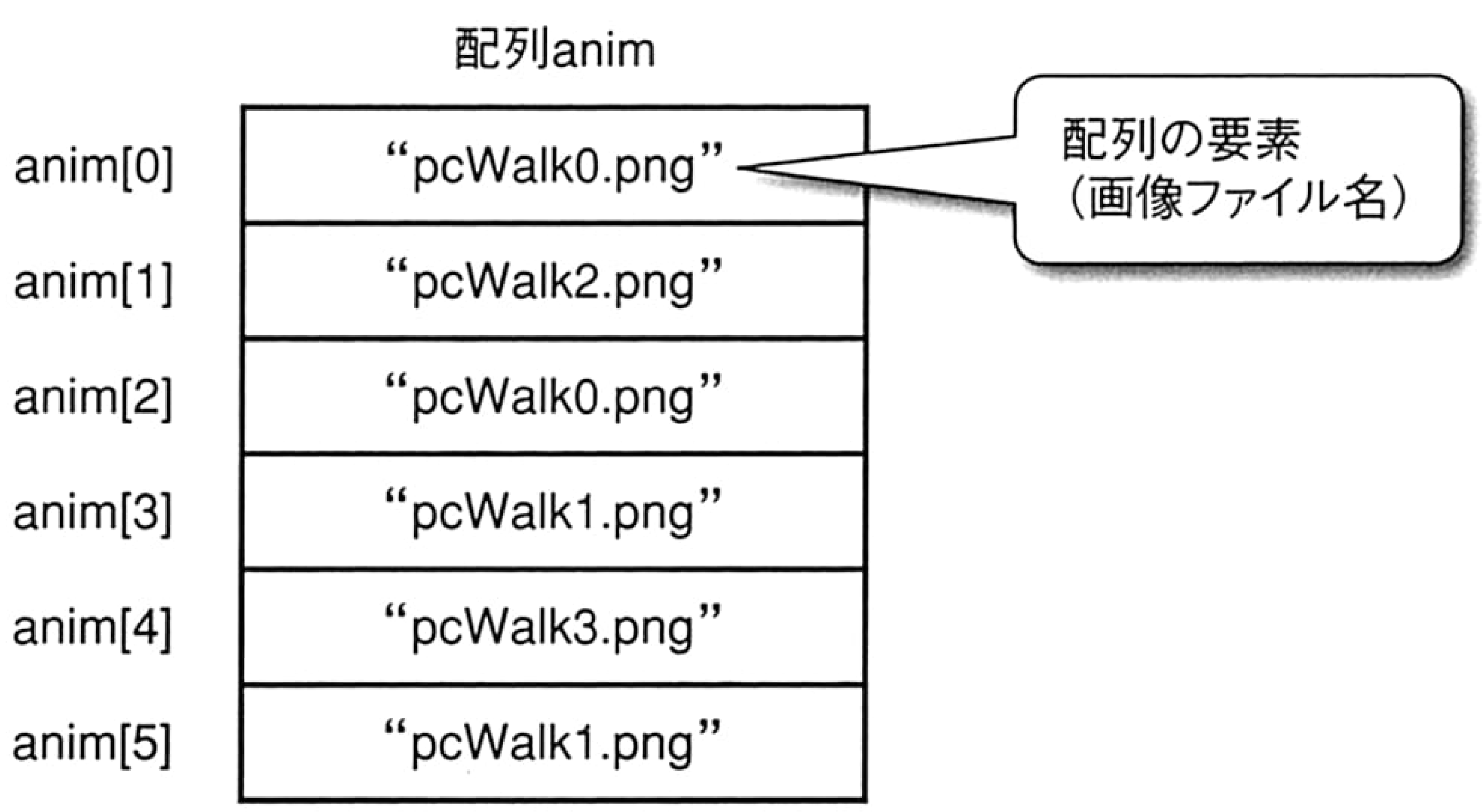


アニメーションを表示する方法

キャラクターが歩くアニメーションを表示する方法について、より具体的に考えてみましょう。アニメーションを行うには、一定時間ごとにキャラクターの画像を入れ替える必要があります。

そこで、まずは画像ファイル名の一覧を作っておきます(Fig.35)。プログラムでは、このような一覧は配列(はいれつ)を使って作るとよいでしょう。配列は複数の値を並べて格納するための機能です。

▼ **Fig.35**: 画像ファイル名の配列



配列には名前を付けることができます。ここではanimという名前にしました。

配列に格納された値を、配列の要素(ようそ)と呼びます。要素には0, 1, 2, ...のように番号が付いています。

配列の各要素を操作するには、

```
配列名[番号]
```

のような記法を使います。例えば、

```
anim[0]
```

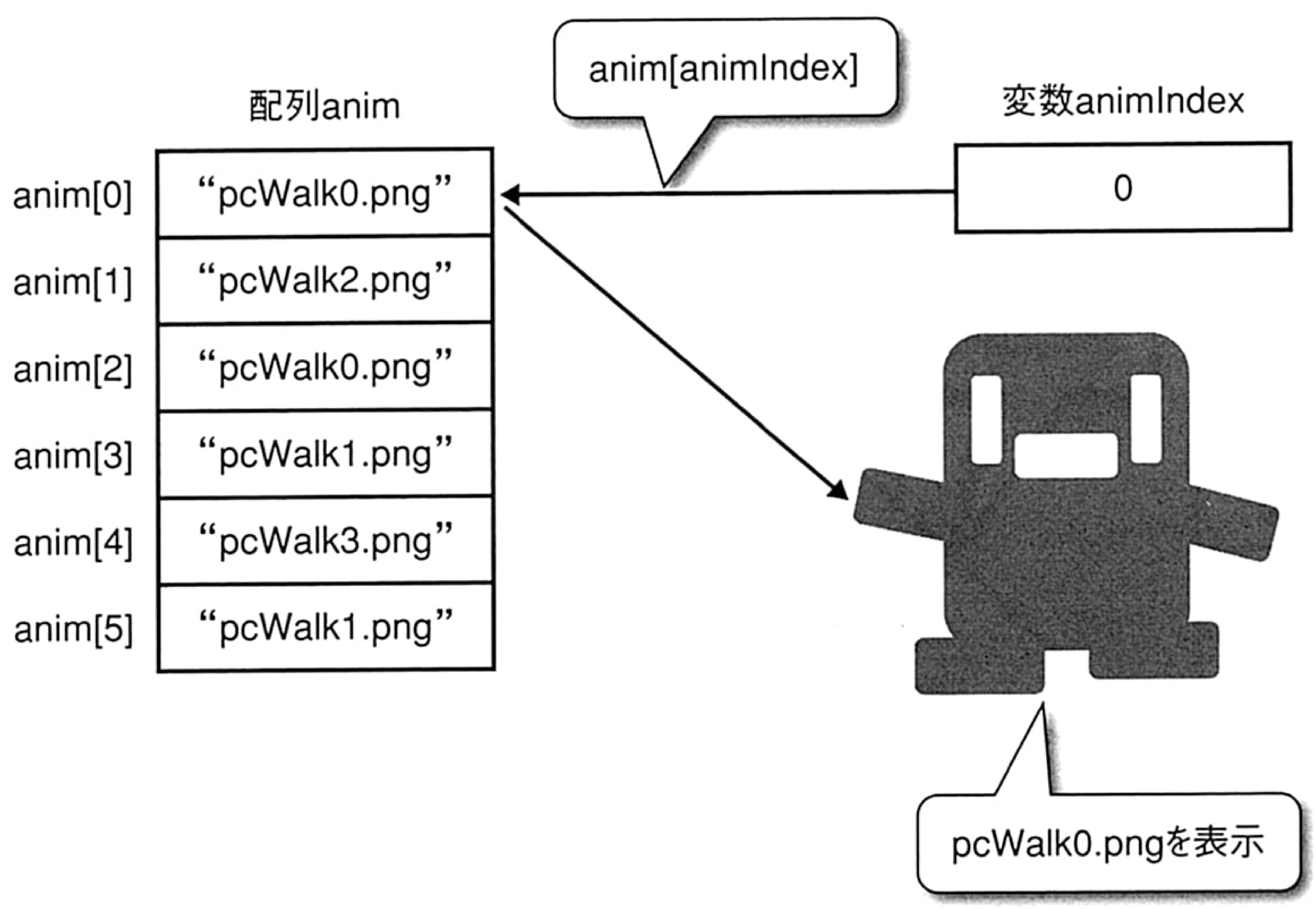
とすると、0番目の要素(pcWalk0.png)を表します。この番号を添字(そえじ)と呼びます。

さて、表示する画像の番号を、変数animIndexで表すことにしましょう。画像ファイル名は、

```
anim[animIndex]
```

のように得ることができます。最初はanimIndexを0にしておくと、anim[0]の画像(pcWalk0.png)を表示することになります(Fig.36)。

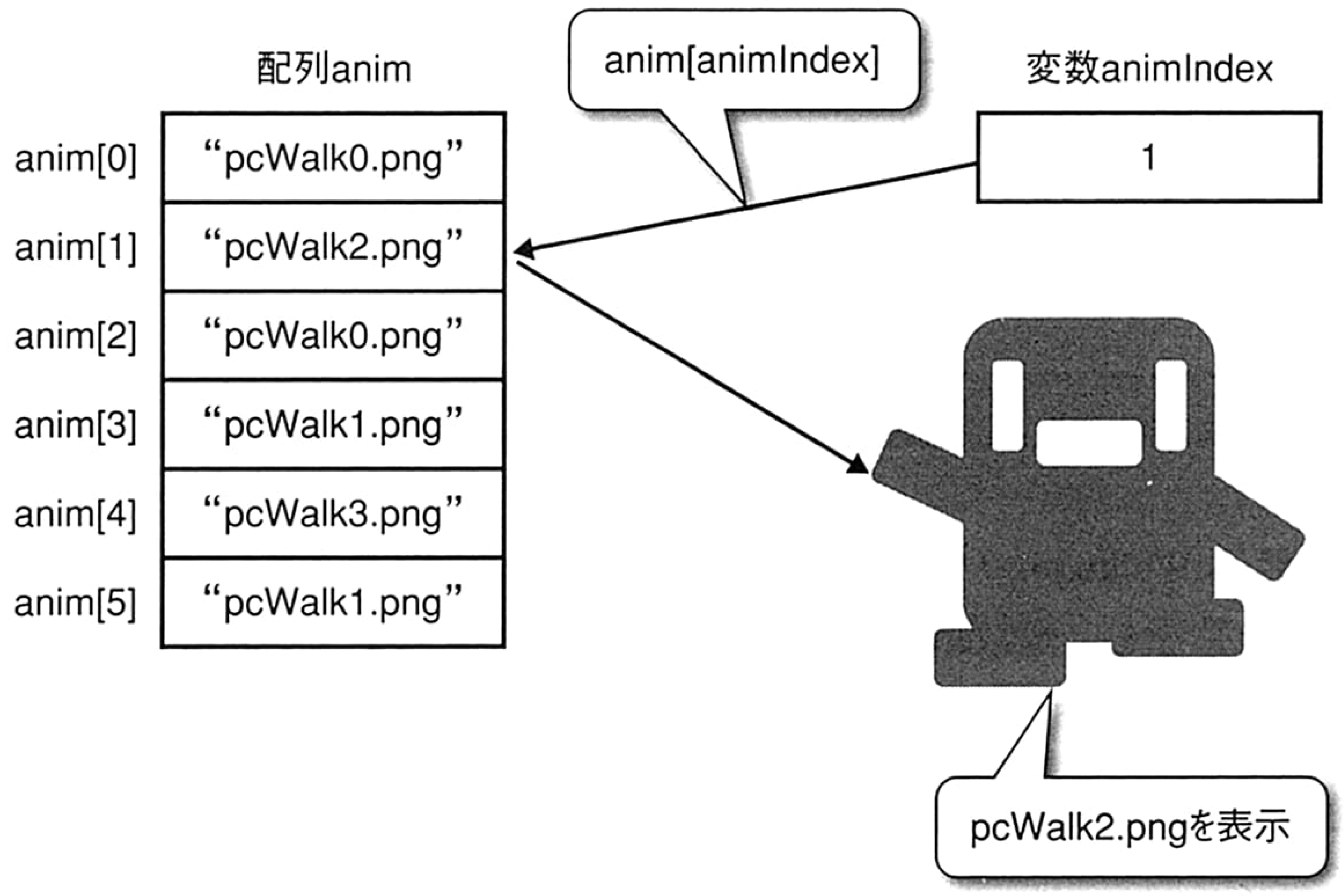
▼ **Fig.36**: 最初は0番目の画像を表示する



アニメーションを行うには、一定時間ごとにanimIndexの値を変化させて、画像を入れ替えます。ここでは適当な短い時間、例えば50ミリ秒(1/20秒)ごとに、animIndexの値に1を加算する

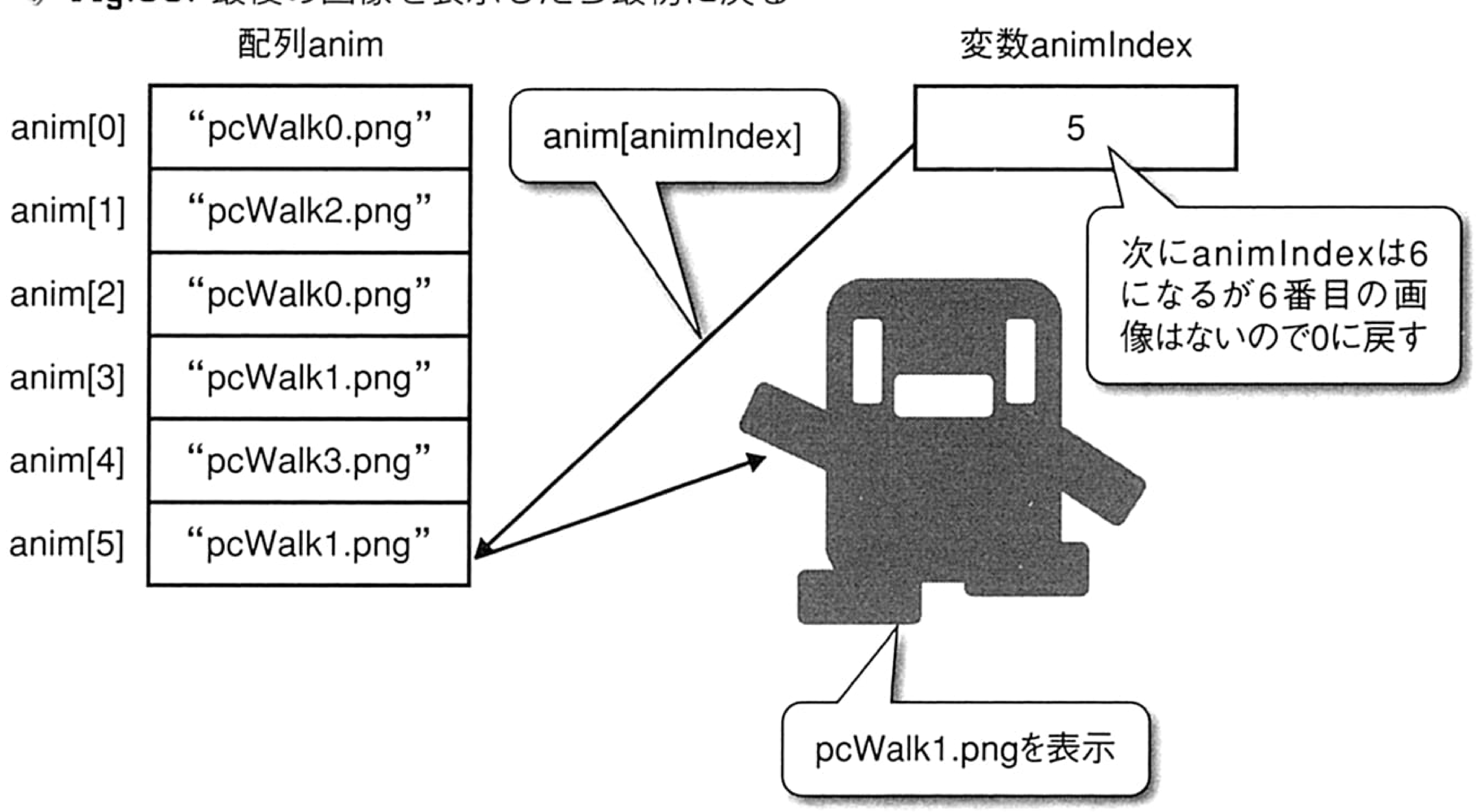
ことにしましょう。animIndexが1になると、anim[1]の画像(pcWalk2.png)が表示されます(Fig.37)。

▼ **Fig.37**: 一定時間後に1番目の画像を表示する



同様に、一定時間が経過するたびにanimIndexの値を更新すれば、2番目の画像(pcWalk0.png)、3番目の画像(pcWalk1.png)を順番に表示することができます。最後に5番目の画像(pcWalk1.png)を表示したら、animIndexは6になります(Fig.38)。6番目以降は画像がないので、ここでanimIndexに0を代入して、0番目の画像に戻ります。

▼ **Fig.38**: 最後の画像を表示したら最初に戻る

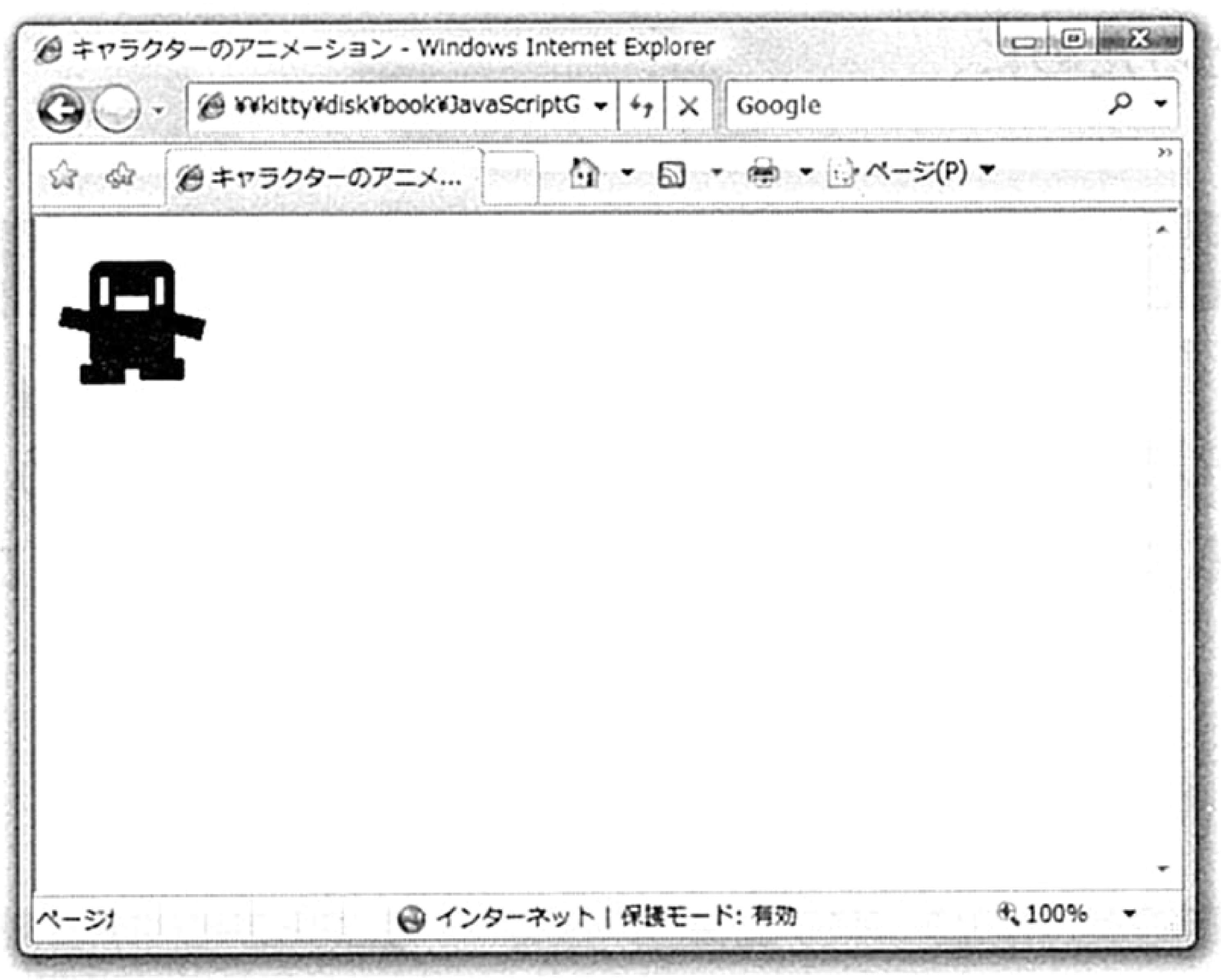


アニメーションを表示するプログラム

検討した方法に沿って、アニメーションを表示するプログラムを作りました(List4)。これはJavaScriptプログラムです。キャラクターを表示するプログラムと同じく、このほかにHTMLファイルが必要です。HTMLファイルの内容は、前回のHTMLファイル(p.20のList2)とほとんど同じなので、JavaScriptプログラムの内容だけを解説します。

JavaScriptファイルはshowAnim.js、HTMLファイルはshowAnim.htmlです。HTMLファイルをブラウザで開けば、プログラムを実行することができます(Fig.39)。画面の左上に、歩くキャラクターのアニメーションが表示されるはずです。

▼ **Fig.39**: 歩くアニメーションの表示



List4は、キャラクターを表示するJavaScriptプログラム(p.24のList3)に、アニメーションの処理を加えたものです。まず、

```
var anim;
var animIndex;
```

のように、2つの変数を追加します。animはアニメーションに使う画像ファイル名の配列で、animIndexは表示する画像の番号です。

animは、

```
anim=new Array(
    "pcWalk0.png", "pcWalk2.png", "pcWalk0.png",
    "pcWalk1.png", "pcWalk3.png", "pcWalk1.png"
);
```

のように初期化します。配列を初期化するためには、この例のように、

```
変数 = new Array(値, 値, 値, ...)
```

と書きます。ここでは6個の画像ファイル名が、配列の0番目から5番目までの要素になります。
最初は0番目の画像(pcWalk0.png)を表示するので、animIndexについては、

```
animIndex=0;
```

のように0で初期化します。
次に、一定時間ごとに画像を入れ替えるために、

```
setInterval(update, 50);
```

として、一定時間ごとにupdate関数を呼び出すように設定します。setInterval関数を使うと、一定時間ごとに実行したい処理を登録することができます。ここでは50ミリ秒(1/20秒)ごとに、update関数を呼び出すことにしました。
update関数では、

```
img.src=anim[animIndex];
```

のように、imgオブジェクトのsrcプロパティ(img要素のsrc属性)を設定して、キャラクターの画像を入れ替えます。最初はanimIndexが0なので、

```
anim[animIndex]
```

は、

```
anim[0]
```

となり、0番目の画像(pcWalk0.png)が表示されます。
次の、

```
animIndex++;
```

では、animIndexに1を加算します。++は1を加算するための記法で、インクリメント演算子(えんざんし)と呼びます。

一定時間(50ミリ秒)が経過すると、再びupdate関数が呼び出されます。今度はanimIndexが1に増えているので、

```
anim[animIndex]
```

は、

```
anim[1]
```

となり、1番目の画像(pcWalk2.png)が表示されます。同様に、一定時間が経過するたびに、2番目の画像(pcWalk0.png)、3番目の画像(pcWalk1.png)が順番に表示されていきます。

最後に5番目の画像(pcWalk1.png)が表示された後には、animIndexは6になります。6番目の画像はないので、0番目に戻す必要があります。そこで、

```
if (animIndex>=anim.length) {
    animIndex=0;
}
```

のようにif文を使って、animIndexが6以上になった場合だけ、animIndexを0に戻します。if文は、

```
if (条件式) {
    処理;
}
```

のように書きます。条件式が成立したときだけ、処理を実行します。

ここで、

```
anim.length
```

は、配列animに格納されている要素の数を表します。このプログラムでは6個の値を格納したので、anim.lengthは6です。そのため、

```
animIndex>=anim.length
```

は、

```
animIndex>=6
```

となり、animIndexが6以上かどうかを調べる条件式になります。animIndexが6以上のときには、

```
animIndex=0;
```

を実行します。つまり、animIndexに0を代入して、0番目の画像に戻ります。

▼ **List4**: アニメーションを表示するJavaScriptプログラム(showAnim.js)

```
// img要素
var img;

// アニメーションの配列
var anim;

// アニメーションの位置
// (現在表示しているアニメーション中の画像の番号)
var animIndex;

// メインルーチン(プログラムの主要な処理)
function main() {

  // キャラクターの画像を表示する
  img=document.createElement("img");
  document.body.appendChild(img);
  img.src="pcStay.png";

  // アニメーションの配列を初期化する
  anim=new Array(
    "pcWalk0.png", "pcWalk2.png", "pcWalk0.png",
    "pcWalk1.png", "pcWalk3.png", "pcWalk1.png"
  );

  // アニメーションの位置を初期化する
```

```
  animIndex=0;

  // 一定時間ごとに実行する処理を登録する
  setInterval(update, 50);
}

// 一定時間ごとに実行する処理
function update() {

  // キャラクターの画像を入れ替える
  img.src=anim[animIndex];

  // アニメーションの位置を更新する
  animIndex++;

  // アニメーションの配列の末尾に到達したら、
  // アニメーションの先頭に戻る
  if (animIndex>=anim.length) {
    animIndex=0;
  }
}
```

配列(はいれつ)

配列は複数の値を格納するための機能です。C/C++/Javaなどのプログラミング言語では、1つの配列には同じ型の値、例えば整数だけや文字列だけを格納します。一方JavaScriptでは、1つの配列に異なる型の値を格納することもできます。
配列を使わなくても、

```
var anim0;
var anim1;
var anim2;
```

のように、多くの変数を並べればよさそうに思えます。しかし配列を使えば、

```
var anim=new Array();
```

のように、変数の定義が簡単になります。また、

```
anim[animIndex]
```

のように、別の変数(animIndex)を使って、配列の何番目の要素を操作したいのかを示すことができます。なおこの［番号］のような記法を添字(そえじ)と呼びます。

演算子(えんざんし)

演算子は各種の演算(計算)を表す記号です。プログラミング言語では、一般的な計算でも用いる加算演算子(+)や減算演算子(−)のほかにも、インクリメント演算子(++)やデクリメント演算子(−−)のように、さまざまな演算子を使うことができます。
JavaScriptには次のような演算子があります。

- 代入演算子(=)
  変数に値を代入する
- 算術演算子(+, −, *, /, %)
  加算・減算・乗算・除算・剰余の演算を行う
- インクリメント演算子(++)
  1を加算する
- デクリメント演算子(−−)
  1を減算する
- 比較演算子(==, !=, <, <=, >, >=, ===, !==)
  等しいかどうかや、大小を調べる
- 論理演算子(&&, ||, !)
  条件式に対して論理積(かつ)・論理和(または)・否定(でない)の演算を行う
- 条件演算子(?:)
  条件式に応じて2つの値のいずれかを返す
- カンマ演算子(,)
  複数の式を順番に計算する
- ビット演算子(&, |, ^, ~, <<, >>, >>>)
  値をビット単位で操作する
- 複合代入演算子(+=, −=, *=, /=, %=, &=, |=, ^=, <<=, >>=, >>>=)
  さまざまな演算と代入をまとめて行う

if文

if文は条件分岐の構文です。条件式に応じて、異なる処理を行うために使います。
条件式が成立すると、条件式の値はtrueになります。成立しないと、値はfalseになります。trueを真(しん)、falseを偽(ぎ)と呼ぶこともあります。if文は、条件式の値がtrueかfalseかに応じて、それぞれ指定された処理を行う構文です。
if文は、

```
if (条件式) {
    処理;
}
```

のように書きます。この記法では、条件式がtrueのときに処理を行います。

また

```
if (条件式) {
    処理A;
}
else {
    処理B;
}
```

のように書くこともできます。この記法では、条件式がtrueのときに処理Aを行い、falseのときに処理Bを行います。
中括弧 {} を書かずに、

```
if (条件式) 処理A;
```

や、

```
if (条件式) 処理A; else 処理B;
```

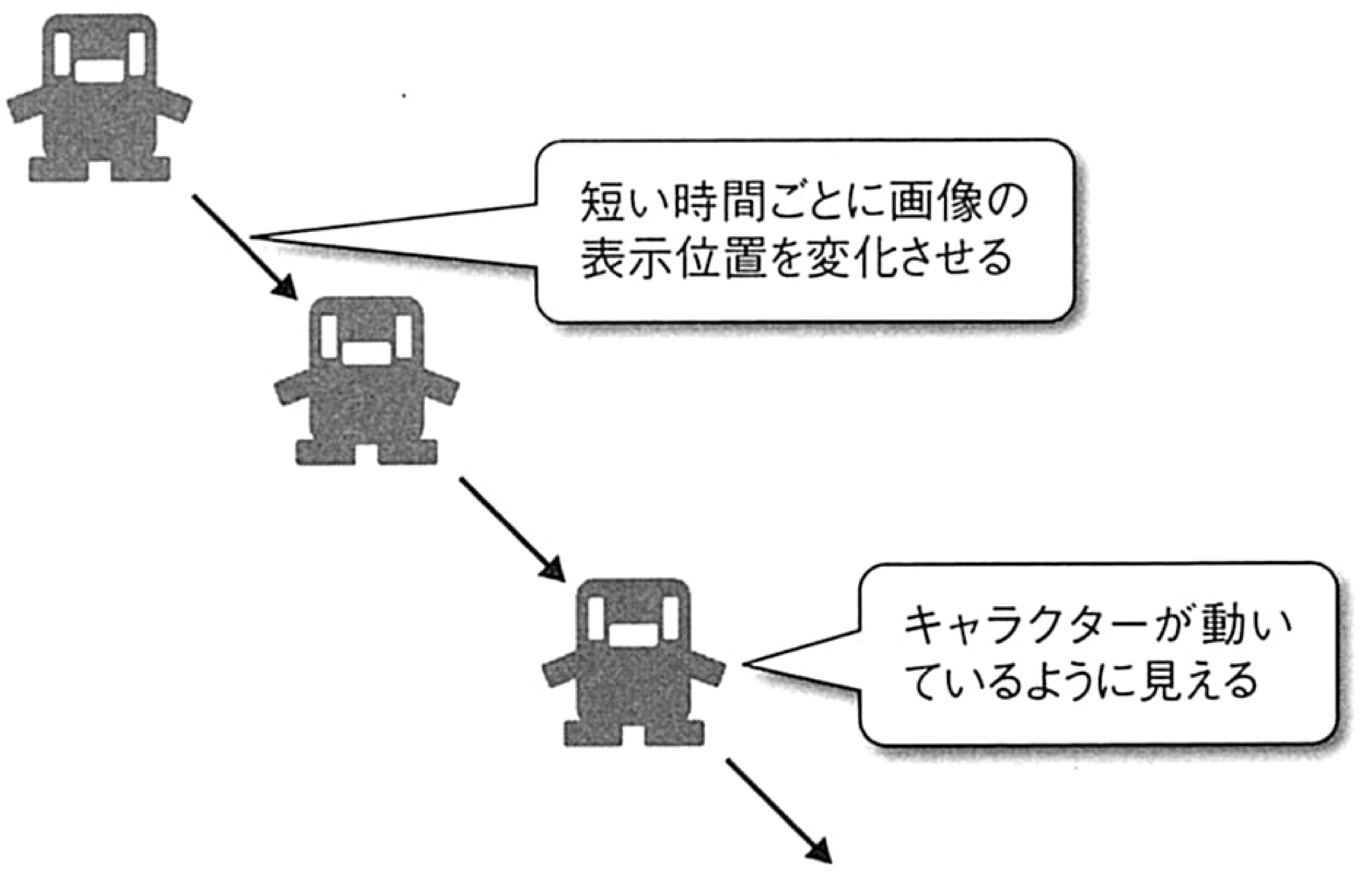
のように記述することもできます。

キャラクターの移動

キャラクターにアニメーションをさせることができましたが、その場でアニメーションするだけではゲームになりません。そこで今度は、キャラクターが画面中を動き回れるようにしましょう。

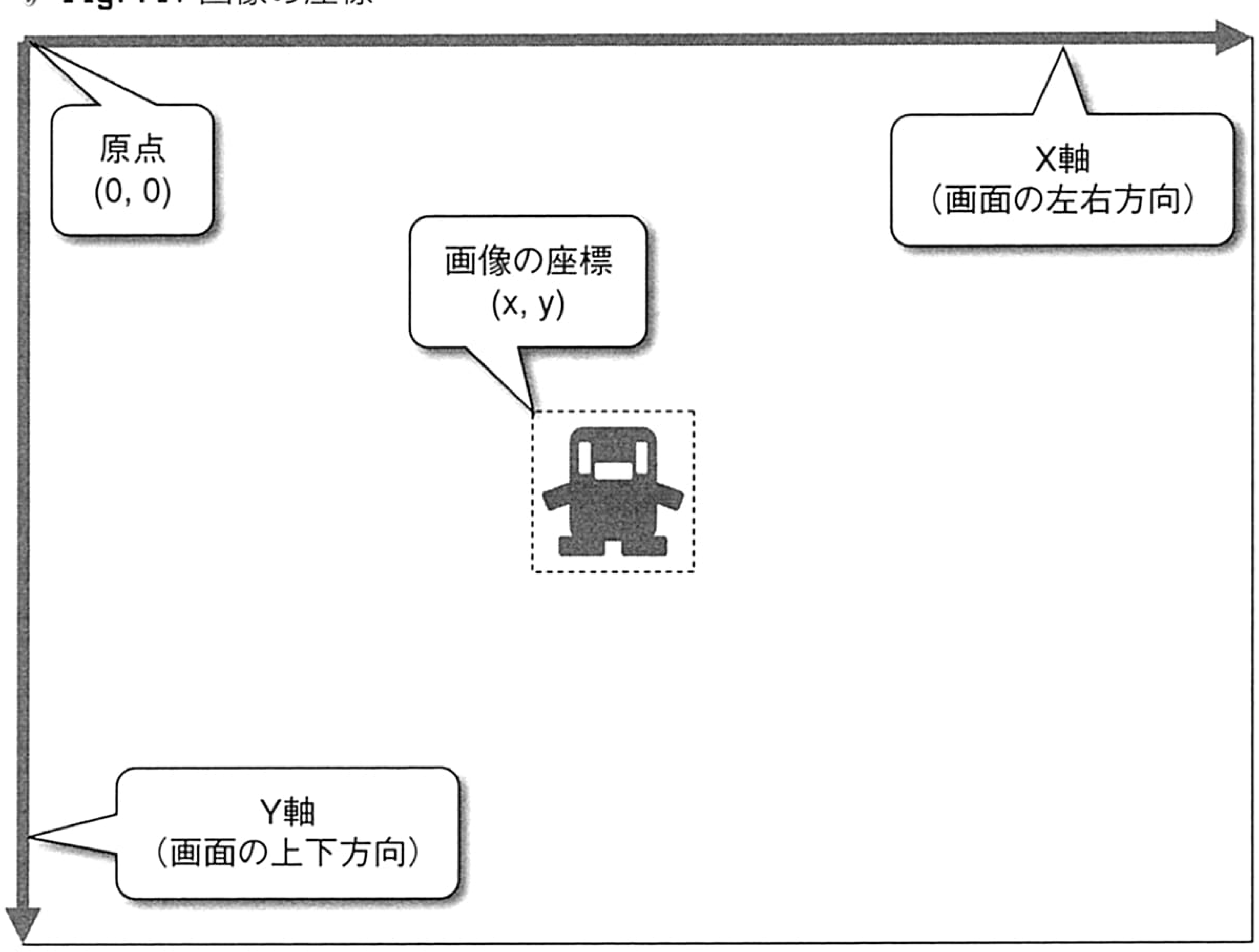
キャラクターの移動は、アニメーションと似た方法で実現できます。アニメーションの場合には、短い時間ごとに画像を入れ替えることによって、キャラクターが動いているように見せます。移動の場合には、短い時間ごとに画像を表示する位置を変化させることによって、キャラクターが滑らかに移動する様子を表現します(Fig.40)。

▼ **Fig.40**: キャラクターを移動させる仕組み



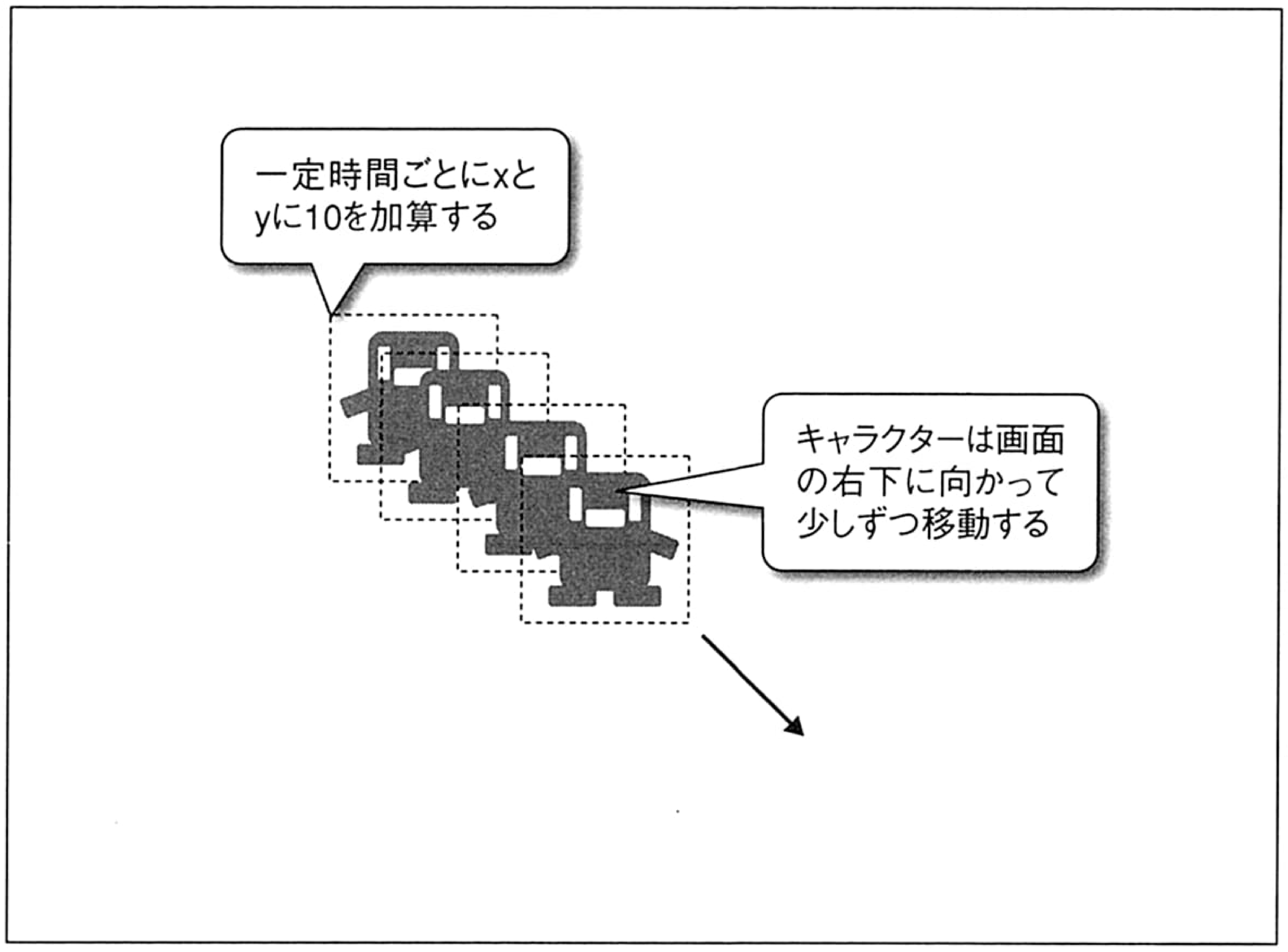
画像を表示する位置は、画面上の座標で指定することができます。座標の指定方法は色々ありますが、画面の左右方向にX軸、上下方向にY軸をとり、(x, y)のような2次元の座標によって指定する方法が一般的です（Fig.41）。座標の原点(0, 0)は画面の左上端、ブラウザの場合にはページの左上端にあります。

▼ **Fig.41**: 画像の座標



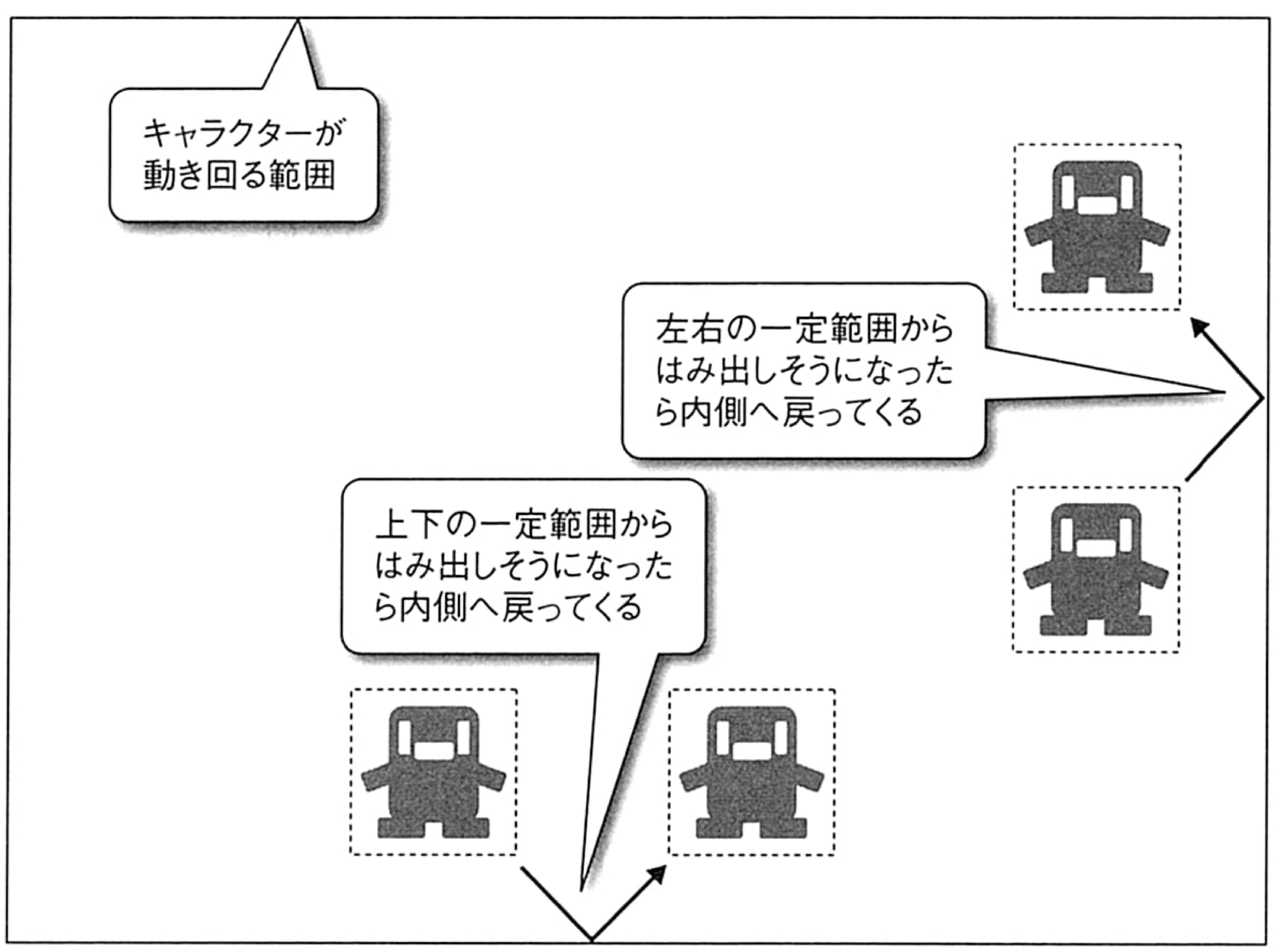
キャラクターの画像を表示する座標を(x, y)としましょう。例えば一定時間ごとに、xとyにそれぞれ10を加算することにします。すると、キャラクターは画面の右下に向かって少しずつ移動します（Fig.42）。

▼ Fig.42: 画像の移動



座標を更新する方法を変えれば、キャラクターにさまざまな動きをさせることができます。例えば、画面上の一定範囲から外れそうになったら、ボールが跳ね返るように内側へ戻ってくるような動きを考えてみましょう(Fig.43)。

▼ Fig.43: 画面上の一定範囲内を動くキャラクター

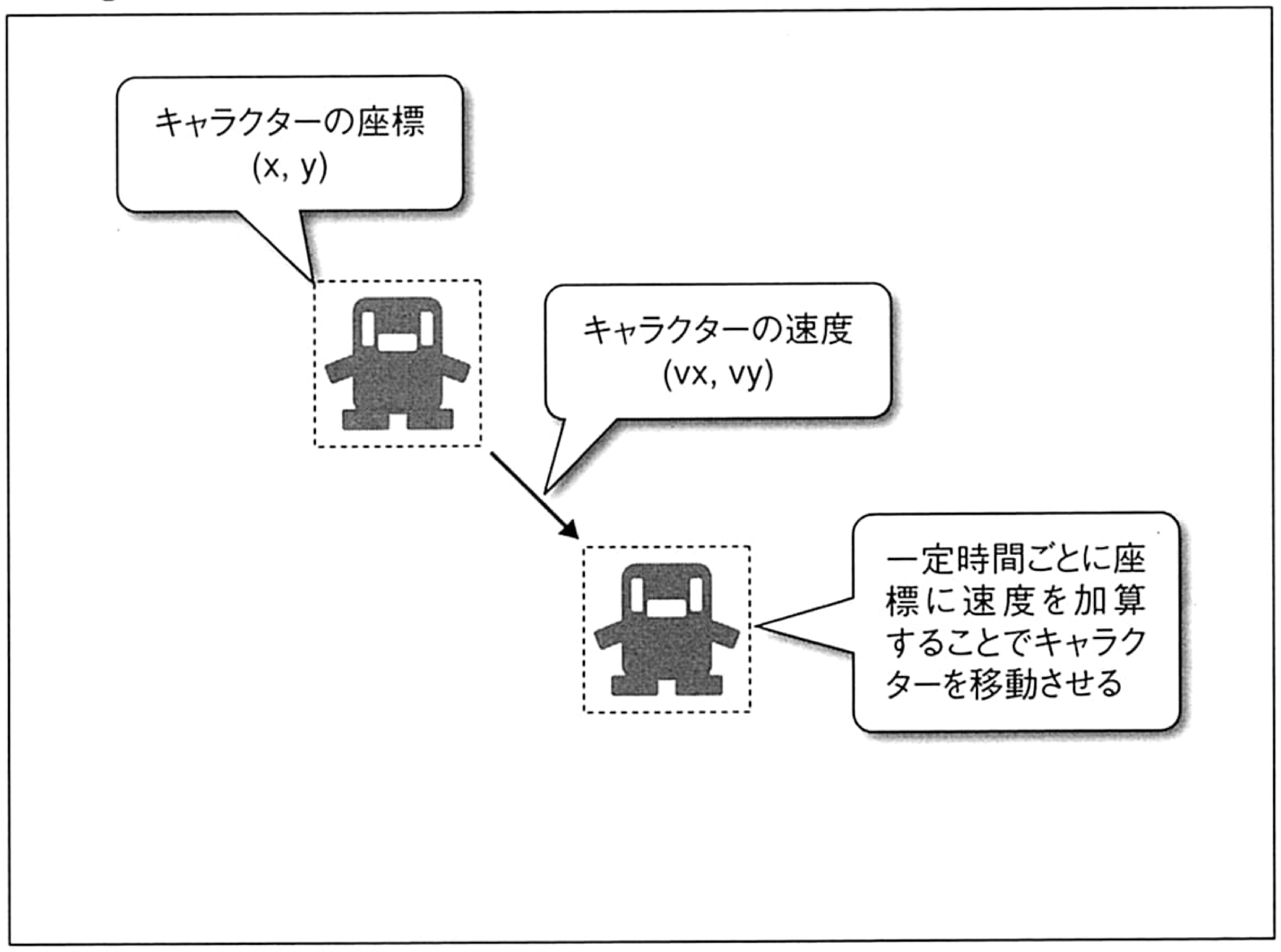


キャラクターの移動速度を(vx, vy)とします。一定時間ごとに、

```
x+=vx;
y+=vy;
```

のように、xにvxを加算し、yにvyを加算することによって、キャラクターを移動させます(Fig.44)。

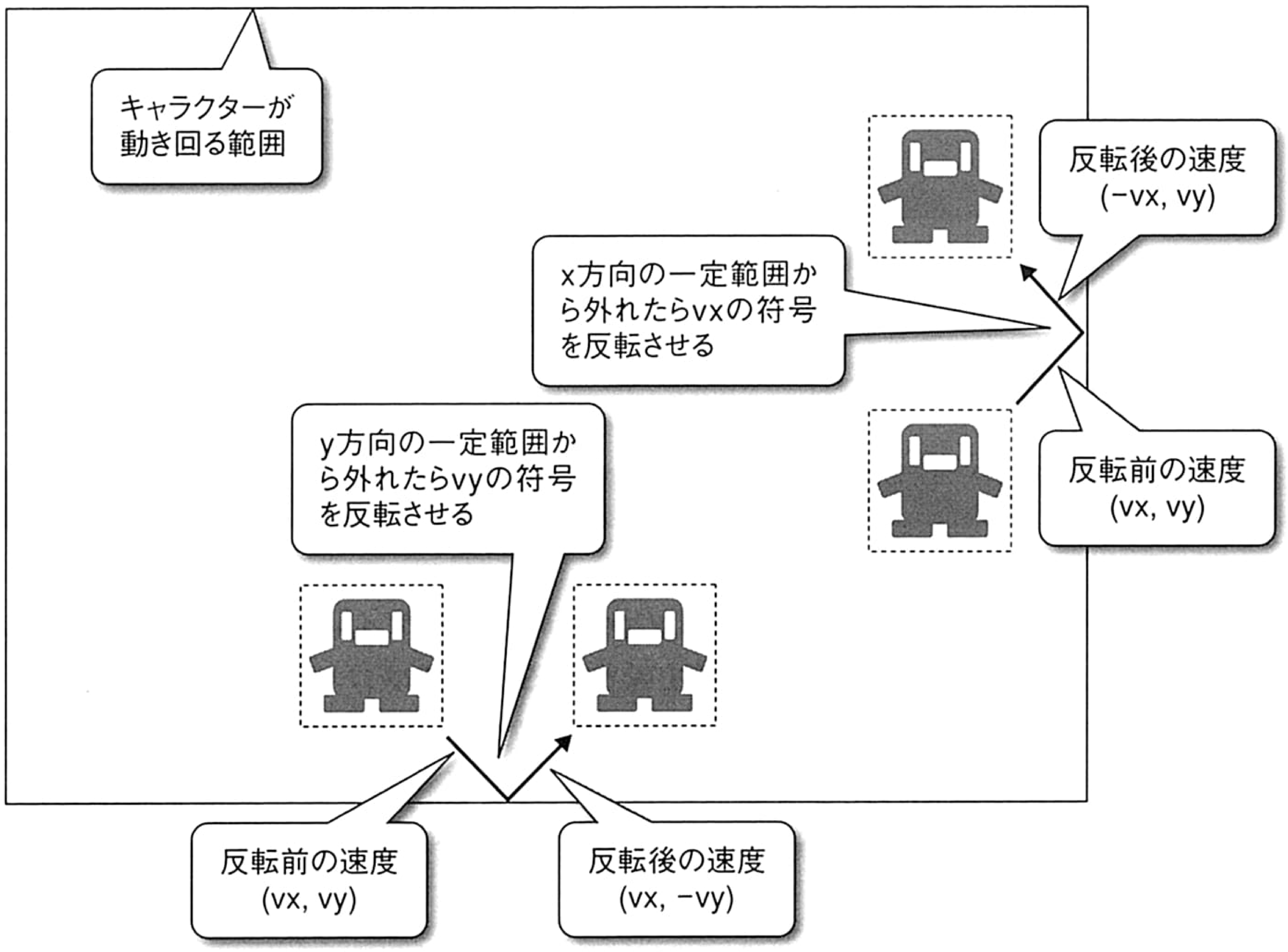
▼ **Fig.44**: 座標に移動速度を加算する



キャラクターが画面上の一定範囲から外れたかどうかは、座標(x, y)を使って調べることができます。xやyが特定の値よりも小さくなったり、大きくなったりしたら、一定範囲から外れたということです。

一定範囲から外れたら、速度の符号を反転させることによって、キャラクターを一定範囲内に戻します(Fig.45)。例えばx方向に外れたらvxの符号を、y方向に外れたらvyの符号を、それぞれ反転させます。これでキャラクターは、ボールが壁で跳ね返るような動きをします。

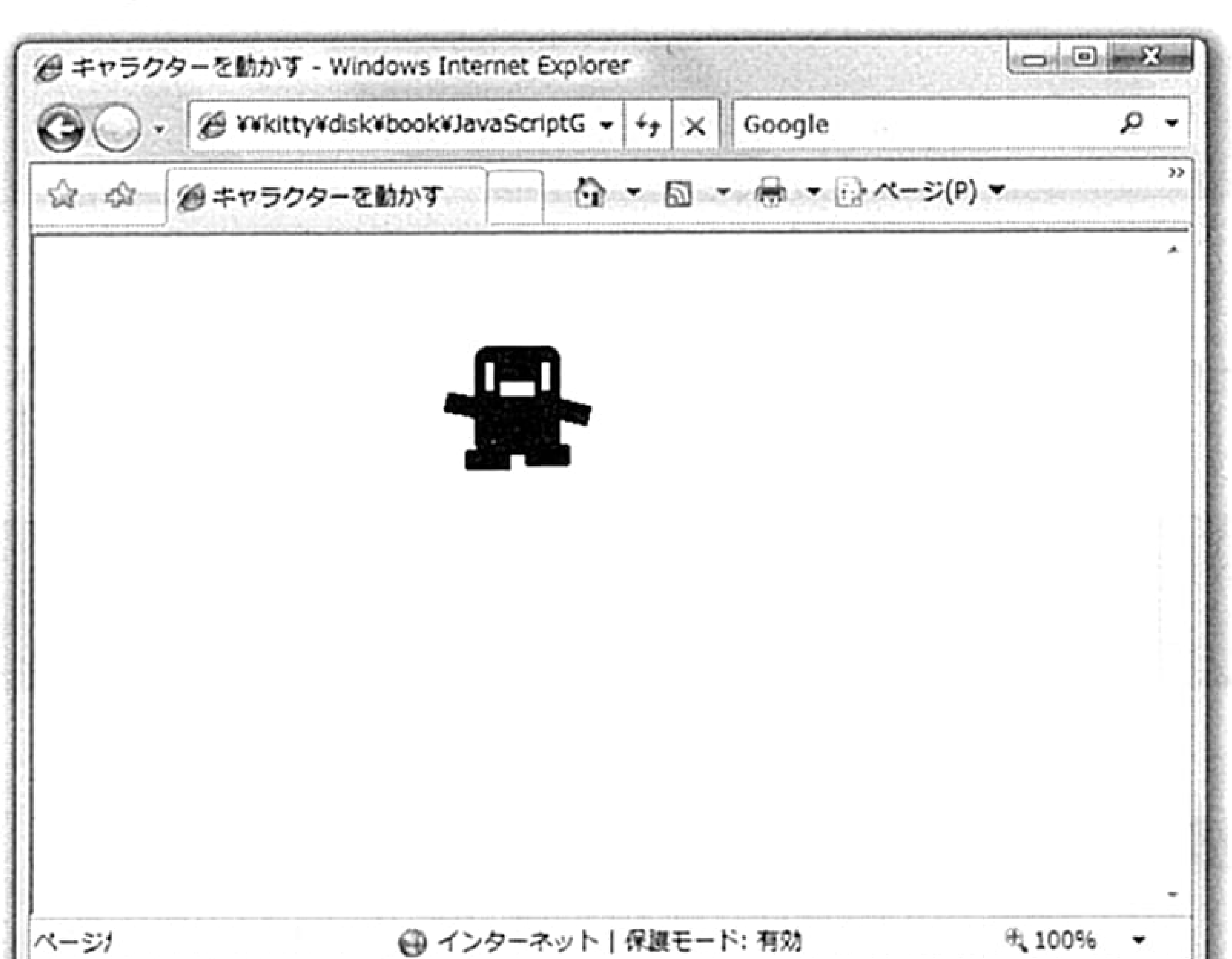
▼ **Fig.45**: 速度の符号を反転させる



キャラクターを移動させるプログラム

以上の方法に沿って、キャラクターを移動させるJavaScriptプログラムを作ります(List5)。HTMLファイル(moveAnim.html)をブラウザで開くと、キャラクターが画面上の一定範囲を動き回る様子が見られます(Fig.46)。

▼ **Fig.46**: 画面上を動き回るキャラクター



List5は、アニメーションを表示するプログラム(p.36のList4)に、キャラクターを移動させる処理を加えたものです。まず、

```
var x, y;
var vx, vy;
```

のように、キャラクターの座標と速度を表す変数を定義します。

メインルーチン(main関数)では、

```
img.style.position="absolute";
```

のように、imgオブジェクトのstyleプロパティのpositionプロパティに、"absolute"を代入します。styleプロパティは、img要素などの表示方法を操作するためのプロパティです。

positionプロパティに"absolute"を代入すると、表示位置を絶対座標で指定することができます。ゲームでは、例えば画面の左上を原点として、(100, 200)の位置に画像を表示したい、といったことがよくあります。そのためには、表示位置を絶対座標で指定する必要があります。

次に、

```
x=0;
y=0;
vx=10;
vy=10;
```

のように、座標と速度を初期化します。座標は画面の左上端に、速度は画面の右下方向になります。

一定時間ごとに実行する処理(update関数)では、

```
x+=vx;
y+=vy;
```

のように、座標に速度を加算することによって、キャラクターの座標を更新します。次に、

```
if (x<=0 || x>=720) vx=-vx;
if (y<=0 || y>=520) vy=-vy;
```

として、キャラクターが画面上の一定範囲から外れたら、速度の符号を反転させます。キャラ

クターは、ページの左上からおよそ800×600ピクセルの範囲を動き回ります。
　最後の、

```
img.style.left=x+"px";
img.style.top=y+"px";
```

では、画像の表示位置を設定します。ここでもstyleプロパティを使います。leftプロパティは画像の左端の座標を、topは画像の上端の座標を表します。
　座標の値に"px"を付けることによって、座標がピクセル単位であることを表します。例えばxが100、yが200のときには、

```
img.style.left="100px";
img.style.top="200px";
```

となります。"px"を付けなくても動くブラウザもありますが(Internet Explorerなど)、動かないブラウザもあるので(Firefoxなど)、必ず付けておくべきです。

▼ **List5**: キャラクターを移動させるプログラム(moveAnim.js)

```
// img要素
var img;

// アニメーションの配列、アニメーションの位置
var anim;
var animIndex;

// キャラクターの座標と速度
var x, y;
var vx, vy;

// メインルーチン
function main() {

  // キャラクターの画像を表示する
  img=document.createElement("img");
  document.body.appendChild(img);
  img.src="pcStay.png";

  // アニメーションを初期化する
  anim=new Array(
    "pcWalk0.png", "pcWalk2.png", "pcWalk0.png",
    "pcWalk1.png", "pcWalk3.png", "pcWalk1.png"
  );
```

```
  animIndex=0;

  // 画像の表示位置を絶対座標で指定する
  img.style.position="absolute";

  // 座標と速度の初期化
  x=0;
  y=0;
  vx=10;
  vy=10;

  // 一定時間ごとに実行する処理を登録する
  setInterval(update, 50);
}

// 一定時間ごとに実行する処理
function update() {

  // アニメーションを再生する
  img.src=anim[animIndex];
  animIndex++;
  if (animIndex>=anim.length) {
    animIndex=0;
  }

  // 座標を更新する
  x+=vx;
  y+=vy;

  // 画面上の一定範囲から外れたら、速度の符号を反転させる
  if (x<=0 || x>=720) vx=-vx;
  if (y<=0 || y>=520) vy=-vy;

  // 画像の座標を更新する
  img.style.left=x+"px";
  img.style.top=y+"px";
}
```

Chapter 2

ライブラリ

共通の処理をライブラリ化する

キャラクターの表示から始めて、アニメーションの表示、キャラクターの移動までを実現することができました。こういった処理は、1つのゲームだけではなく、色々なゲームに共通です。こういった共通の処理を使いやすくライブラリにまとめておけば、ゲームを素早く開発することができます。

本書のサンプルでは、共通の処理をcommon.jsというJavaScriptファイルにまとめました。このライブラリには次のような機能があります。

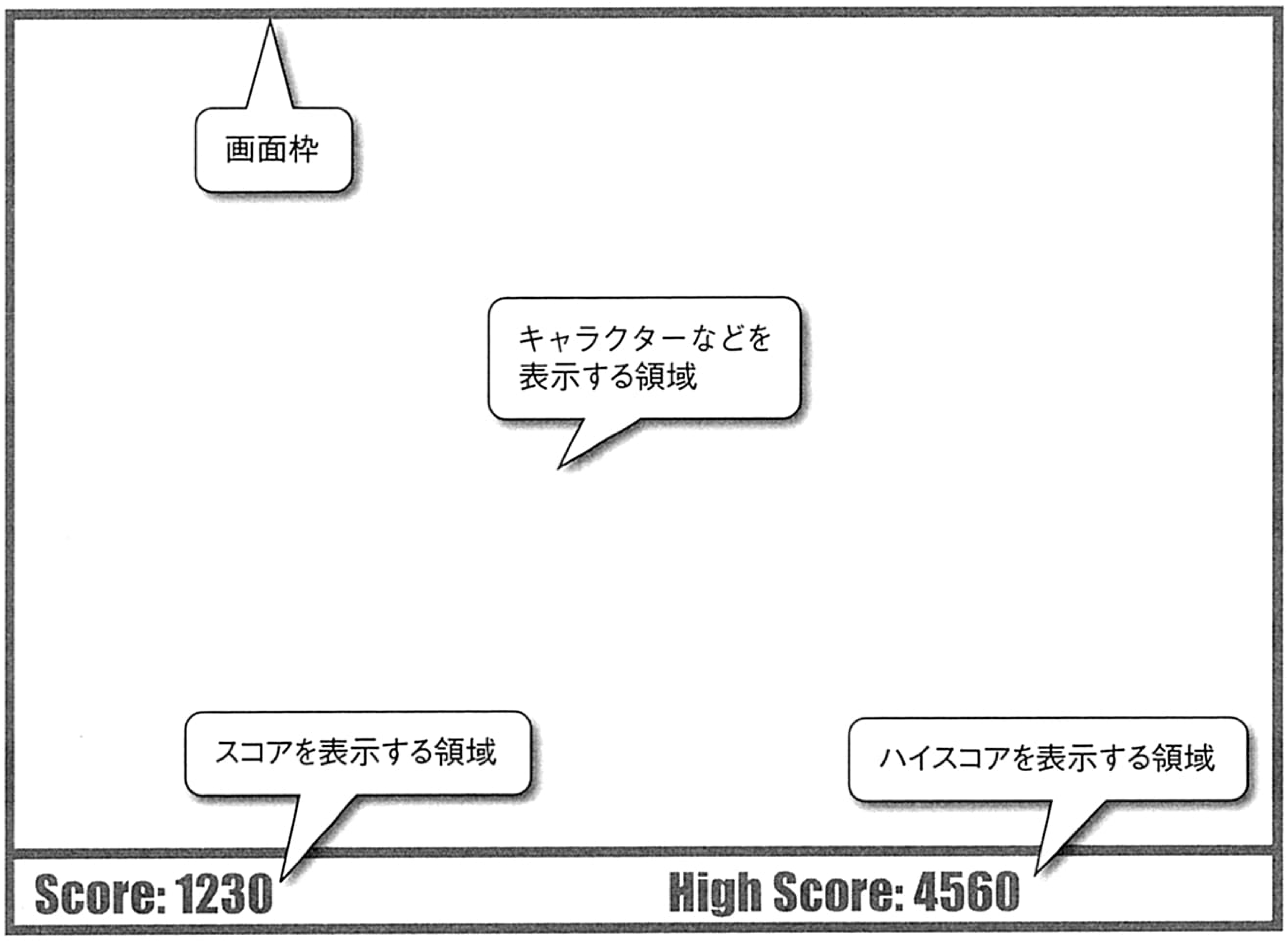
- ゲームで使う定数の定義
- 乱数の生成
- デバッグメッセージの表示
- スプライト(キャラクターなどの表示・アニメーション・移動)
- フィールド(背景の表示)
- スプライトやフィールドの当たり判定処理(接触したかどうかの判定)
- ゲーム画面の表示
- スコアの管理
- キーボードやマウスからの入力
- ゲーム全体の制御
- 通信

詳細な内容は、これから少しずつ解説していきます。このライブラリを使うと、本書のサンプルのようなゲームを簡単に作ることができます。また、このライブラリを参考にして、独自のライブラリを開発するのにも役立てていただけるでしょう。

ゲーム画面を作る

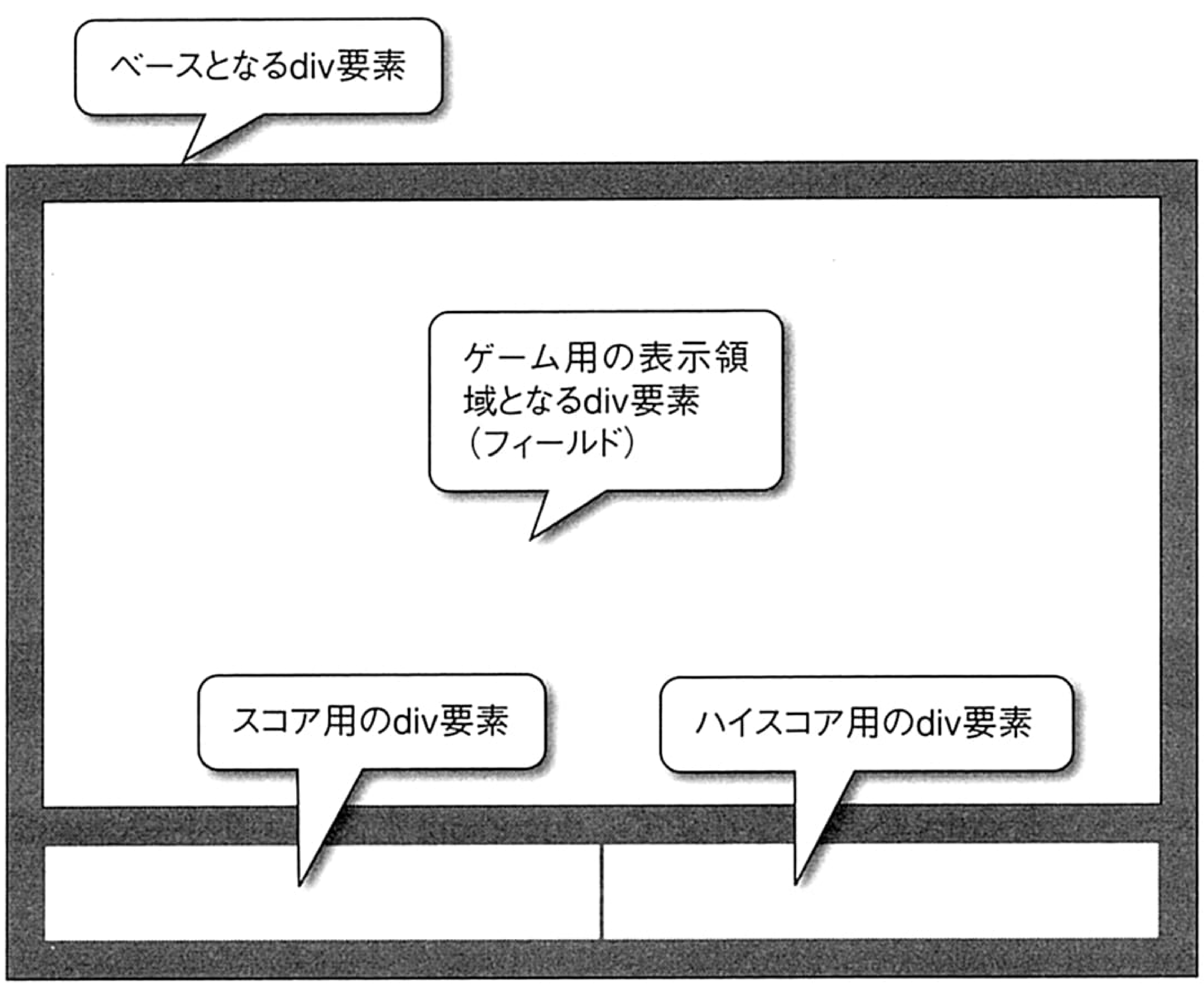
ここまでのサンプルでは、何もない真っ白な画面にキャラクターを表示していました。しかしゲームとなると、移動できる範囲の境界を表す画面枠や、スコアを表示する領域などが必要です(Fig.1)。

▼ Fig.1: ゲーム画面に必要なもの



本書ではHTMLのdiv要素を使って、このようなゲーム画面を構成することにしました(Fig.2)。div要素はページ上にブロックを作るための要素です。div要素の中には、他のさまざまな要素(img要素など)を配置することができます。

▼ Fig.2: ゲーム画面の構成



まず、ベースとなる大きなdiv要素を配置します。ベースは黒色にしました。その上に重ねて、ゲーム用の表示領域となるdiv要素を配置します。この領域を、本書ではフィールドと呼ぶことにします。フィールドは白色にしました。

フィールドの下には、スコアとハイスコア(最高記録のスコア)を表示するためのdiv要素を配置します。これらも白色にしました。

ゲーム画面を作るプログラム

ゲーム画面を作るプログラムは、ライブラリ(common.js)に含まれています(List1)。ライブラリには他にも多くの機能がありますが、ここではゲーム画面を作る部分だけを抜粋して紹介します。

List1では、

```
function Game() {
```

のように、Game関数を定義します。実は、これはGameオブジェクトの定義になっています。

JavaScriptではさまざまなオブジェクトを利用します。これまでのサンプルでは、documentオブジェクト、bodyオブジェクト、imgオブジェクトなどを使いました。これらのオブジェクトは、JavaScript側があらかじめ用意しているオブジェクトです。

一方、プログラマが必要に応じて、自分のプログラムで独自のオブジェクトを定義することもできます。そこでList1では、独自のGameオブジェクトを定義しています。

オブジェクトを定義する方法はいくつかありますが、ここでは関数の定義を利用して、オブジェクトを定義しています。まず、

```
var obj=this;
```

のように、変数objにthisを代入しています。このthisは、thisキーワードと呼ばれ、オブジェクト自身を指すという特別な機能があります。ここでは、thisはGameオブジェクト自身を指します。

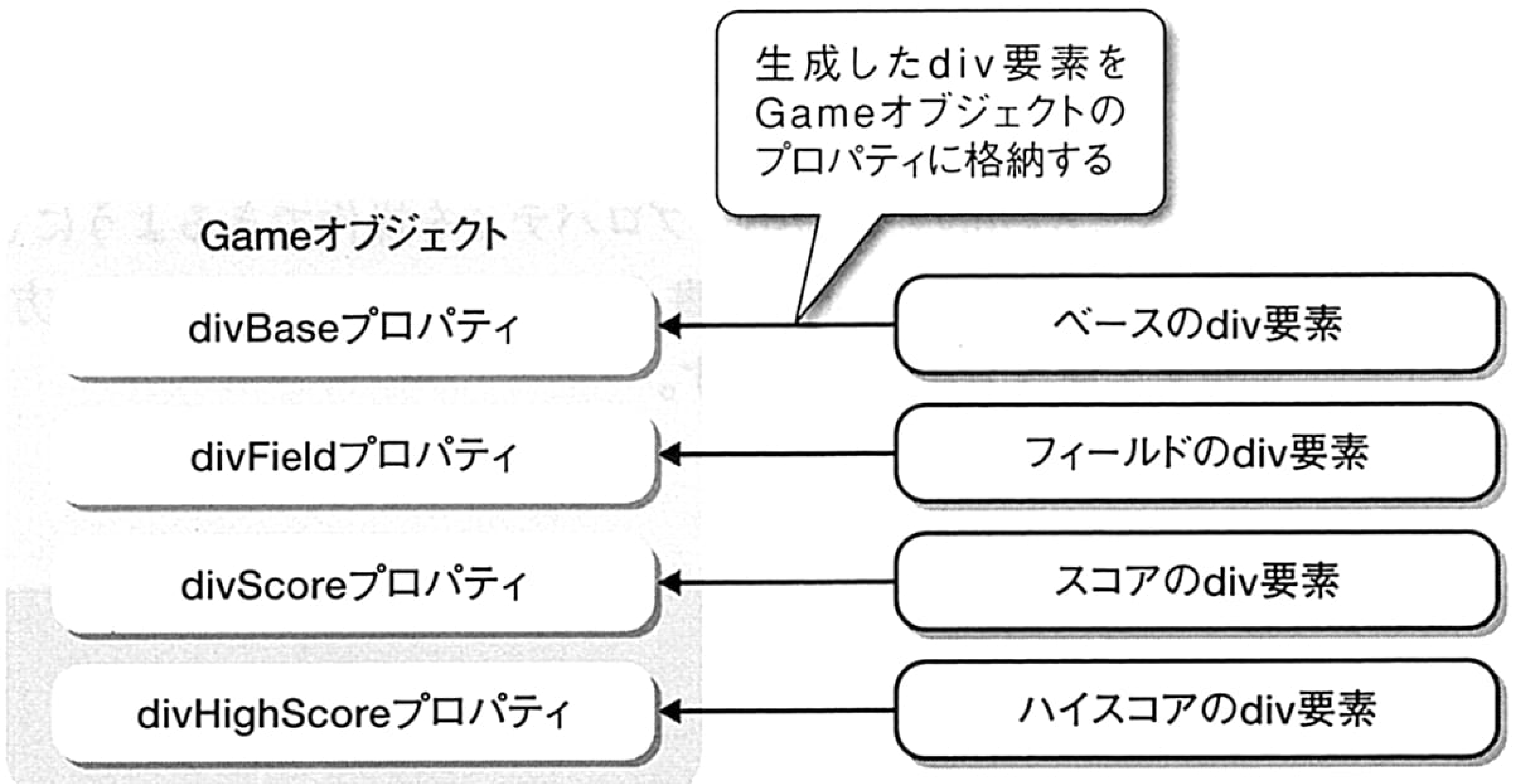
次に、

```
obj.divBase=document.createElement("div");
```

として、ゲーム画面のベースとなるdiv要素を作ります。documentオブジェクトのcreateElement関数は、img要素の作成にも利用しました(p.23)。

ここでは生成したdiv要素を、objオブジェクトのdivBaseプロパティに格納します。objにはthisを代入したので、objはGameオブジェクト自身を指します。つまりここでは、生成したdiv要素を、GameオブジェクトのdivBaseプロパティに格納しているわけです(Fig.3)。

▼ **Fig.3**: 生成したdiv要素をプロパティに格納する



同様に、フィールドやスコア・ハイスコアについても、

```
obj.divField=document.createElement("div");
obj.divScore=document.createElement("div");
obj.divHighScore=document.createElement("div");
```

のようにdiv要素を生成し、それぞれGameオブジェクトのdivField, divScore, divHighScoreプロパティに代入します。これらのプロパティは、後でフィールドに画像を表示したり、スコアやハイスコアを更新したりするときに使います。

生成したdiv要素を画面に表示するには、appendChild関数を使う必要があります。ベースのdiv要素については、

```
document.body.appendChild(obj.divBase);
```

のように、ボディ要素に追加します。フィールドやスコア・ハイスコアについては、

```
obj.divBase.appendChild(obj.divField);
obj.divBase.appendChild(obj.divScore);
obj.divBase.appendChild(obj.divHighScore);
```

のように、ベースのdiv要素に追加します。これで、ベースの内部にフィールド・スコア・ハイスコアを配置することができます。

次に各々のdiv要素について、位置・サイズ・色などを設定します。設定方法は同じなので、ベースのdiv要素について説明しましょう。まず、

```
style=obj.divBase.style;
```

のように、変数styleを使って、ベースのdiv要素のstyleプロパティを操作できるようにします。この処理は必須ではありませんが、obj.divBase.styleと書くよりも、styleとだけ書いた方がすっきりするので行いました。with文を使う方法もあります。

div要素の表示位置については、

```
style.position="absolute";
style.left=0+"px";
style.top=0+"px";
```

のように、絶対座標で指定することにして、座標は(0, 0)にしました。

色については、

```
style.backgroundColor=BORDER_COLOR;
```

のように、背景色を設定します。定数BORDER_COLORは"black"(黒色)です。

サイズについては、

```
style.width=FIELD_WIDTH+BORDER_SIZE*2+"px";
style.height=FIELD_HEIGHT+SCORE_HEIGHT+BORDER_SIZE*3+"px";
```

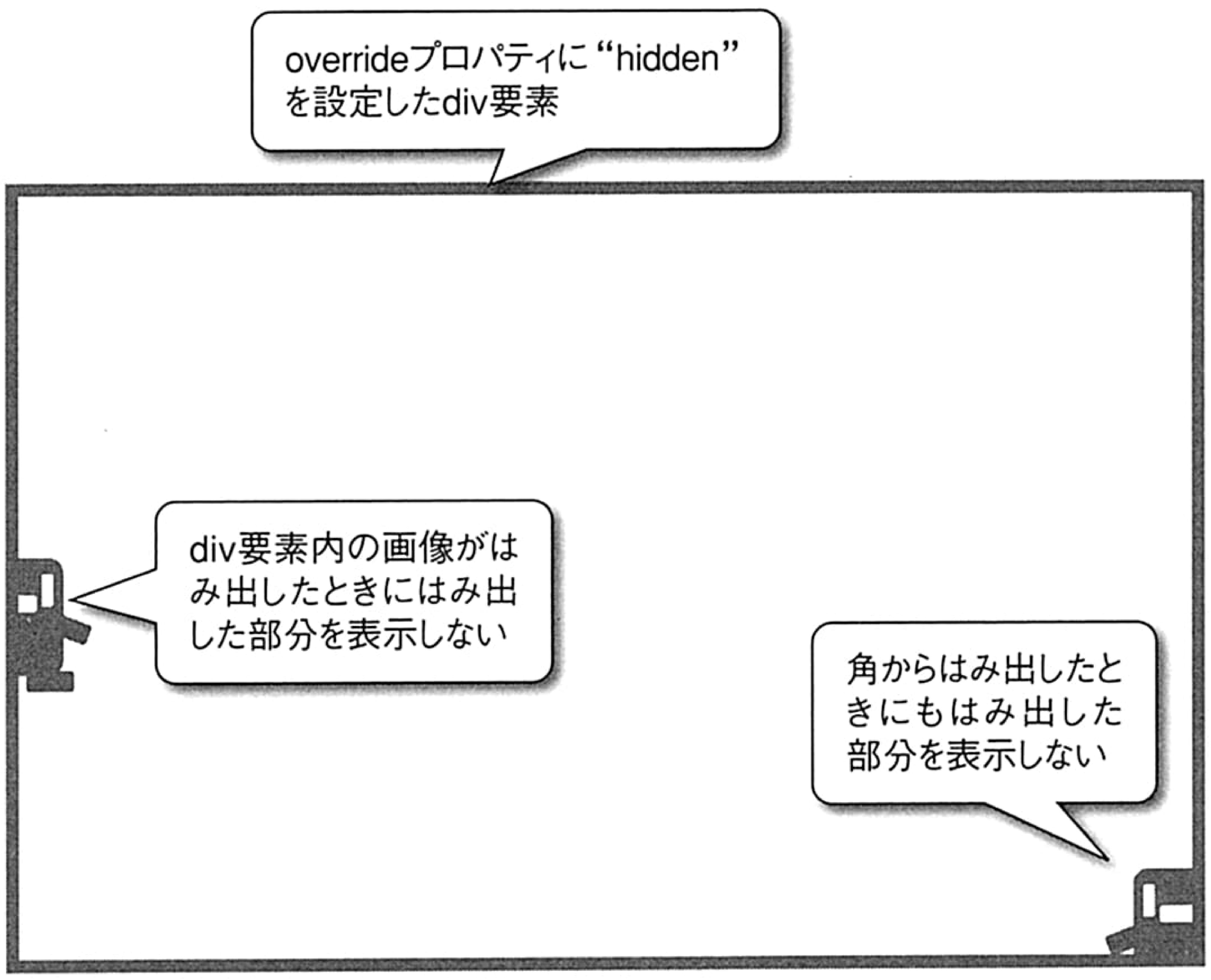
としました。定数の詳細はList1をご覧ください。幅が970ピクセル、高さが615ピクセルになります。サンプルを見やすくするために、ブラウザで表示するページとしては少し大きめにしました。実際にゲームを作るときには、プレイヤーの動作環境を考慮して、大きさを調整するとよいでしょう。

フィールド・スコア・ハイスコアについても、ベースと同様の設定を行います。フィールドについては、

```
style.overflow="hidden";
```

のように、overflowプロパティを設定します。これはフィールドからキャラクターなどがはみ出したときに、はみ出した部分を表示しないための処置です。div要素のoverflowプロパティに"hidden"を設定すると、要素内に配置した他の要素(img要素)などがはみ出したときに、はみ出した部分を表示しなくなります(Fig.4)。"hidden"を設定しないと、はみ出した部分が表示されてしまったり、スクロールバーが出てしまったりします。

Fig.4: フィールドからはみ出した部分を非表示にする



また、スコアとハイスコアについては、

```
style.fontFamily=SCORE_FONT_FAMILY;
style.fontSize=SCORE_FONT_SIZE;
```

のように、フォントの種類とサイズも指定します。スコアの表示については、第4章で詳しく説明します。

List1: ゲーム画面を作るプログラム(common.js)

```
// 定数
var CELL_SIZE=80;
var CELL_XCOUNT=12;
var CELL_YCOUNT=7;
var FIELD_WIDTH=CELL_SIZE*CELL_XCOUNT;
var FIELD_HEIGHT=CELL_SIZE*CELL_YCOUNT;
```

```
var FIELD_COLOR="white";
var BORDER_SIZE=5;
var BORDER_COLOR="black";
var SCORE_HEIGHT=40;
var SCORE_FONT_FAMILY="Impact, sans-serif";
var SCORE_FONT_SIZE="24pt";

// ゲーム本体のオブジェクト
function Game() {

  // 変数objがオブジェクト自身を指すようにする
  var obj=this;

  // 関数内で一時的に使う変数の定義
  var style;
  var i;

  // ベース表示領域
  obj.divBase=document.createElement("div");
  document.body.appendChild(obj.divBase);
  style=obj.divBase.style;
  style.position="absolute";
  style.backgroundColor=BORDER_COLOR;
  style.left=0+"px";
  style.top=0+"px";
  style.width=FIELD_WIDTH+BORDER_SIZE*2+"px";
  style.height=FIELD_HEIGHT+SCORE_HEIGHT+BORDER_SIZE*3+"px";

  // フィールド表示領域
  obj.divField=document.createElement("div");
  obj.divBase.appendChild(obj.divField);
  style=obj.divField.style;
  style.overflow="hidden";
  style.position="absolute";
  style.backgroundColor=FIELD_COLOR;
  style.left=BORDER_SIZE+"px";
  style.top=BORDER_SIZE+"px";
  style.width=FIELD_WIDTH+"px";
  style.height=FIELD_HEIGHT+"px";

  // スコア表示領域
  obj.divScore=document.createElement("div");
  obj.divBase.appendChild(obj.divScore);
  style=obj.divScore.style;
  style.overflow="hidden";
  style.position="absolute";
  style.backgroundColor=FIELD_COLOR;
```

```
  style.left=BORDER_SIZE+"px";
  style.top=FIELD_HEIGHT+BORDER_SIZE*2+"px";
  style.width=FIELD_WIDTH/2+"px";
  style.height=SCORE_HEIGHT+"px";
  style.fontFamily=SCORE_FONT_FAMILY;
  style.fontSize=SCORE_FONT_SIZE;

  // ハイスコア表示領域
  obj.divHighScore=document.createElement("div");
  obj.divBase.appendChild(obj.divHighScore);
  style=obj.divHighScore.style;
  style.overflow="hidden";
  style.position="absolute";
  style.backgroundColor=FIELD_COLOR;
  style.left=BORDER_SIZE+FIELD_WIDTH/2+"px";
  style.top=FIELD_HEIGHT+BORDER_SIZE*2+"px";
  style.width=FIELD_WIDTH/2+"px";
  style.height=SCORE_HEIGHT+"px";
  style.fontFamily=SCORE_FONT_FAMILY;
  style.fontSize=SCORE_FONT_SIZE;

  // ...(中略)...
}
```

with文

指定したオブジェクトのプロパティを、簡潔に操作するための機能です。with文は、

```
with (オブジェクト) {
    処理;
}
```

のように書きます。処理では、オブジェクト名を省略してプロパティを操作することができます。例えば、

```
obj.divBase.style.position="absolute";
obj.divBase.style.backgroundColor="black";
```

は、

```
with (obj.divBase.style) {
    position="absolute";
    backgroundColor="black";
}
```

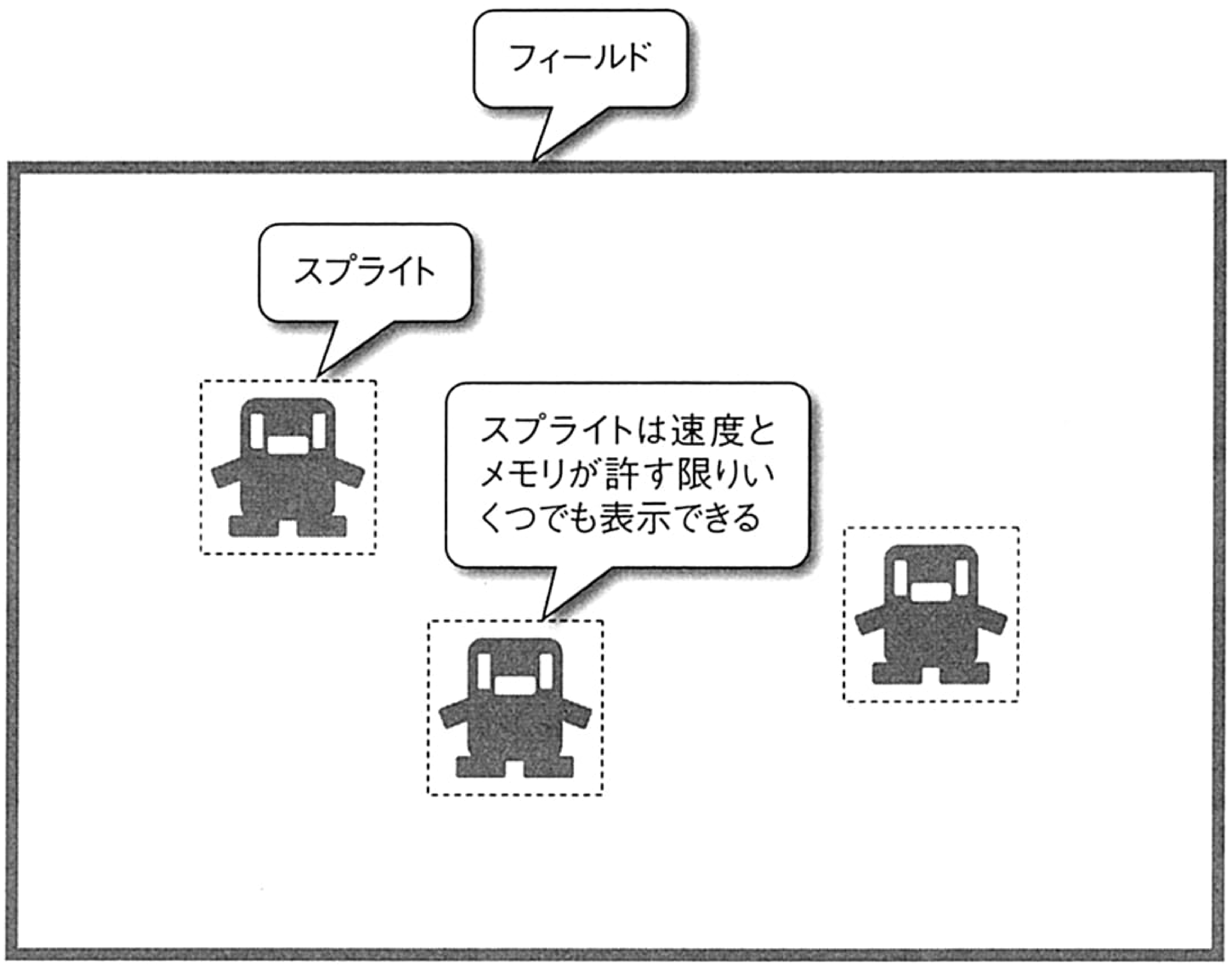
のように書けます。

スプライト

ゲーム画面を作る処理は、Gameオブジェクトにまとめました。同様に次は、キャラクターの表示·アニメーション·移動に関する処理を、オブジェクトにまとめることを考えましょう。このオブジェクトを、本書ではスプライト(Sprite)と呼ぶことにします。スプライトというのは、ゲームにおいて画面に画像を表示するための機能の1つです。

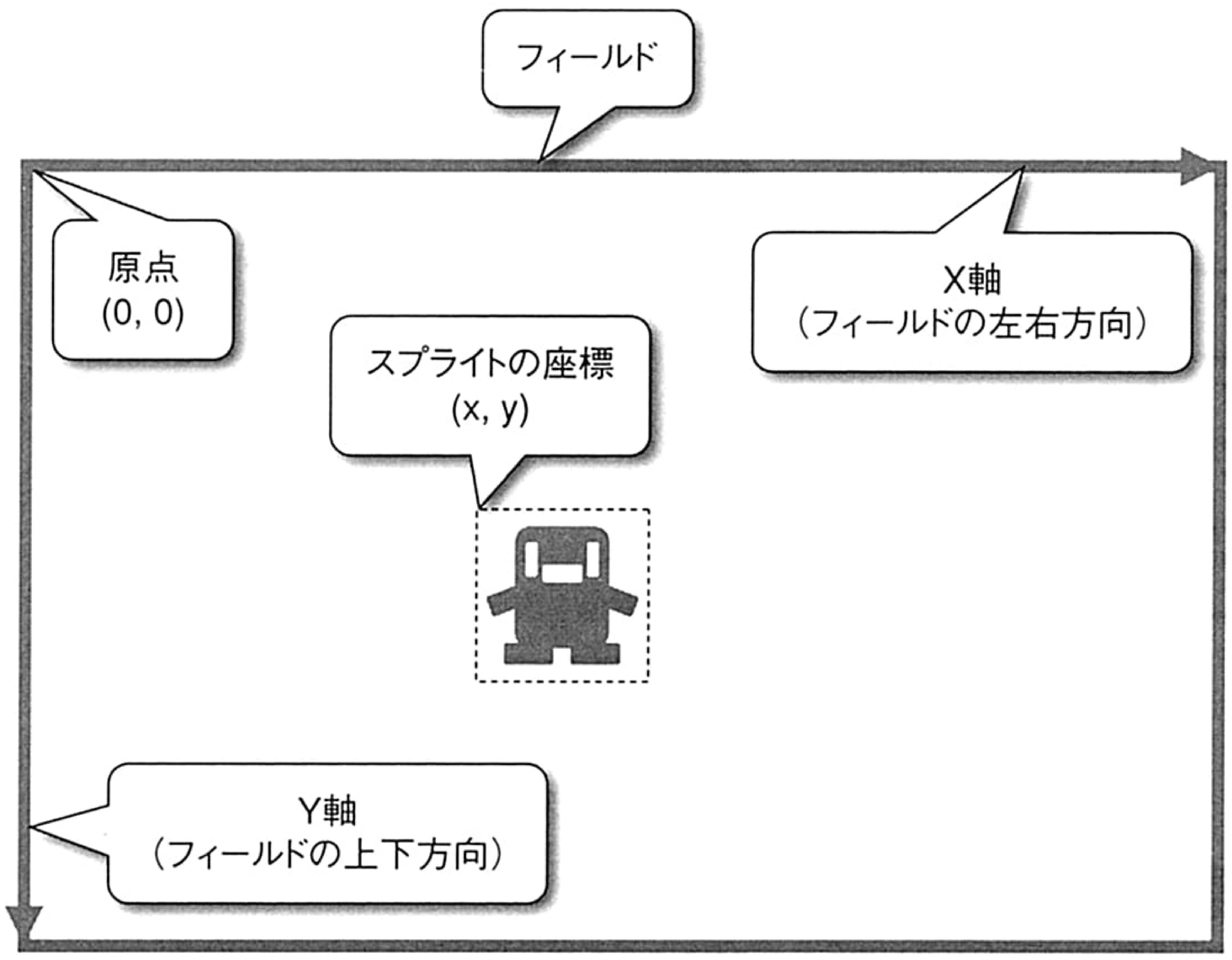
本書では、スプライトをゲーム画面のフィールド領域に表示することにします(Fig.5)。スプライトはマシンの速度とメモリが許す限り、いくつでも表示できます。

▼ **Fig.5**: スプライトの表示



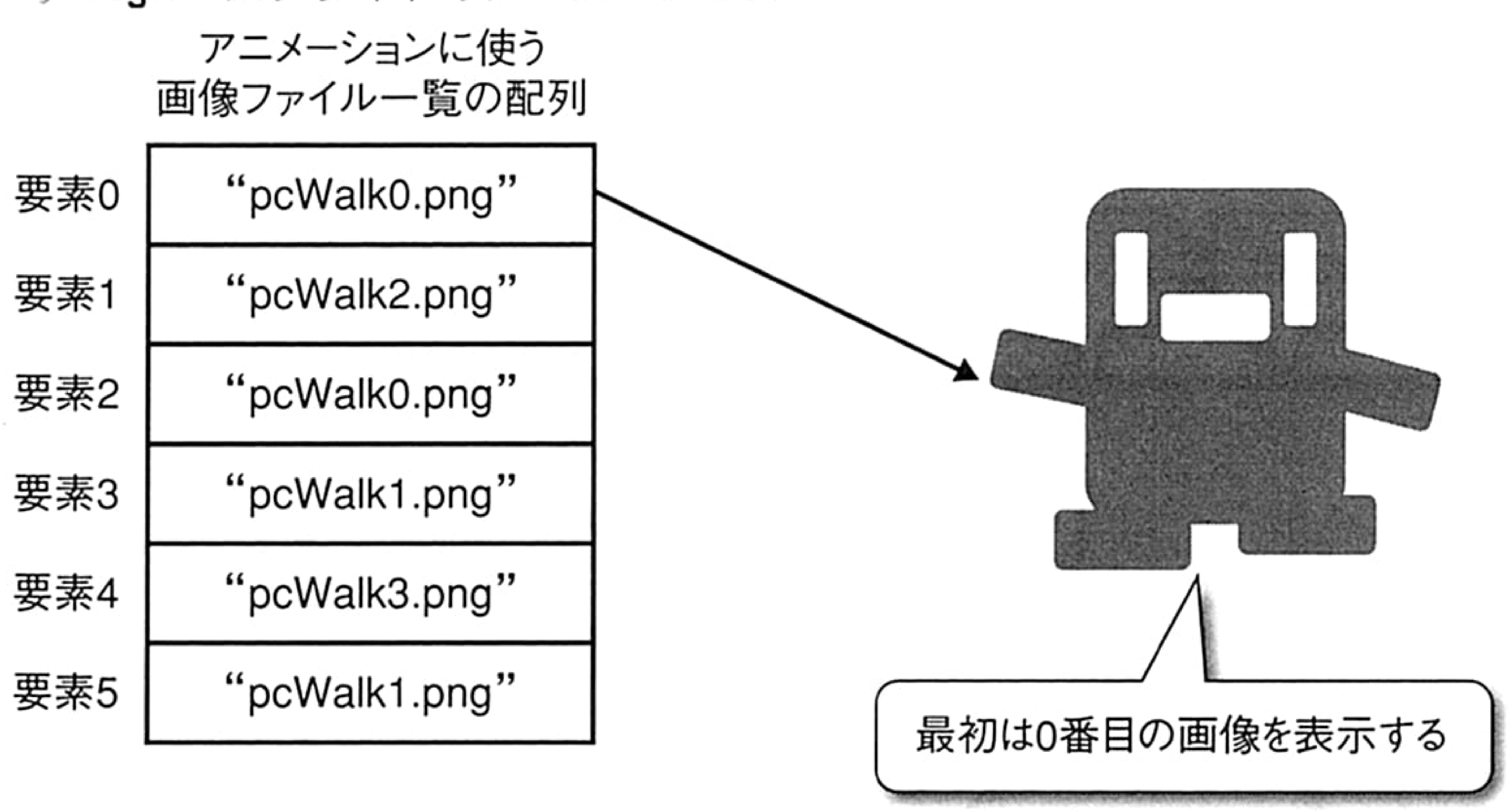
スプライトの位置は、フィールドの左上端を(0, 0)として、X座標とY座標で指定することにします(Fig.6)。一定時間ごとに座標を変化させれば、スプライトを移動させることができます。

▼ **Fig.6**: スプライトの位置



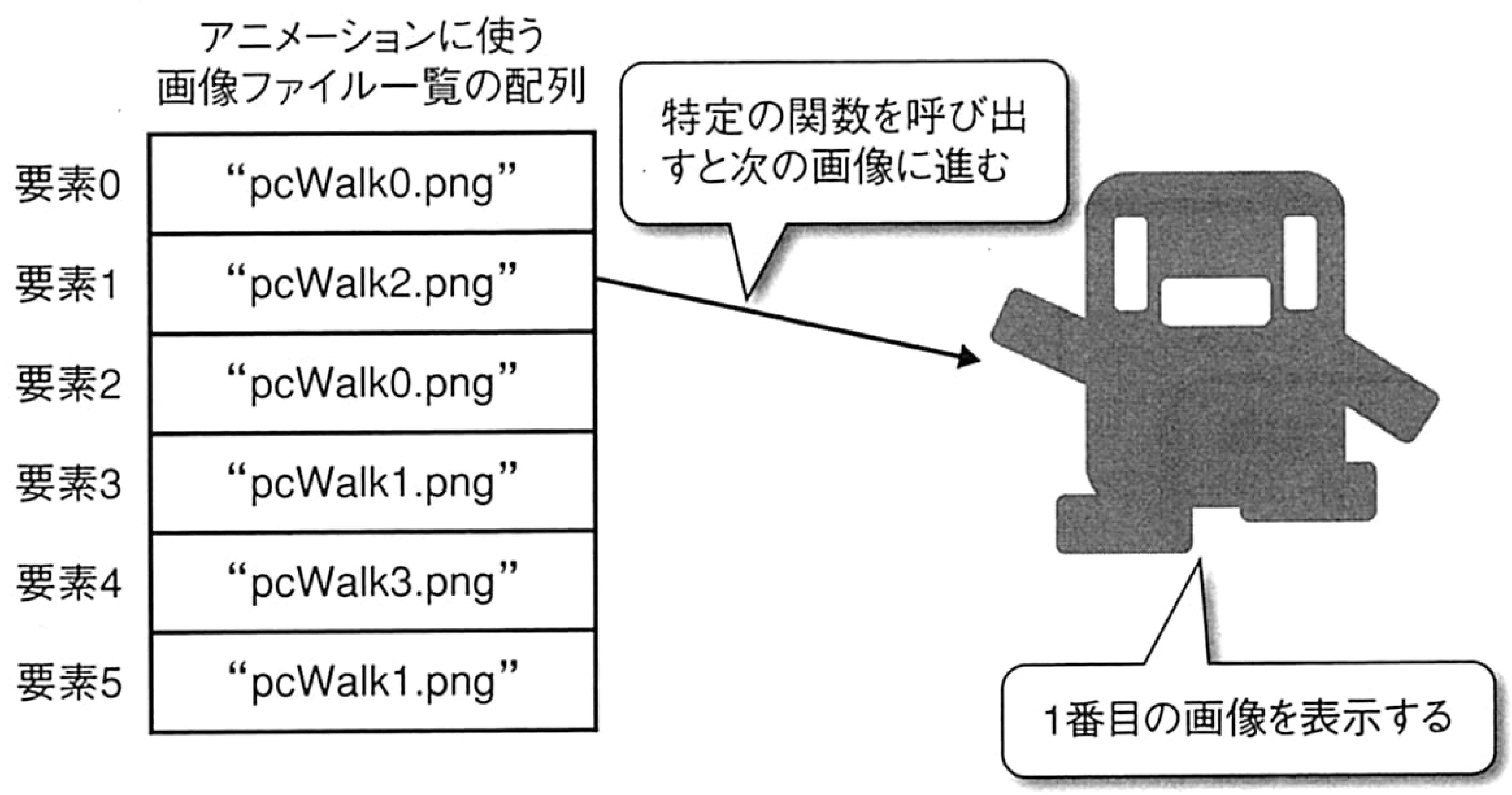
スプライトはアニメーションの表示も行います(Fig.7)。アニメーションに使う画像ファイル名の一覧を配列に格納しておき、スプライトに登録します。最初にスプライトは、登録された画像の中から、0番目の画像を表示します。

▼ **Fig.7**: スプライトのアニメーション



特定の関数を呼び出したら、スプライトが次の画像を表示するようにします(Fig.8)。一定時間ごとに関数を呼び出せば、アニメーションが表示できます。アニメーションの末尾に到達したら、自動的に先頭へ戻るようにします。

▼ Fig.8: アニメーションを再生する

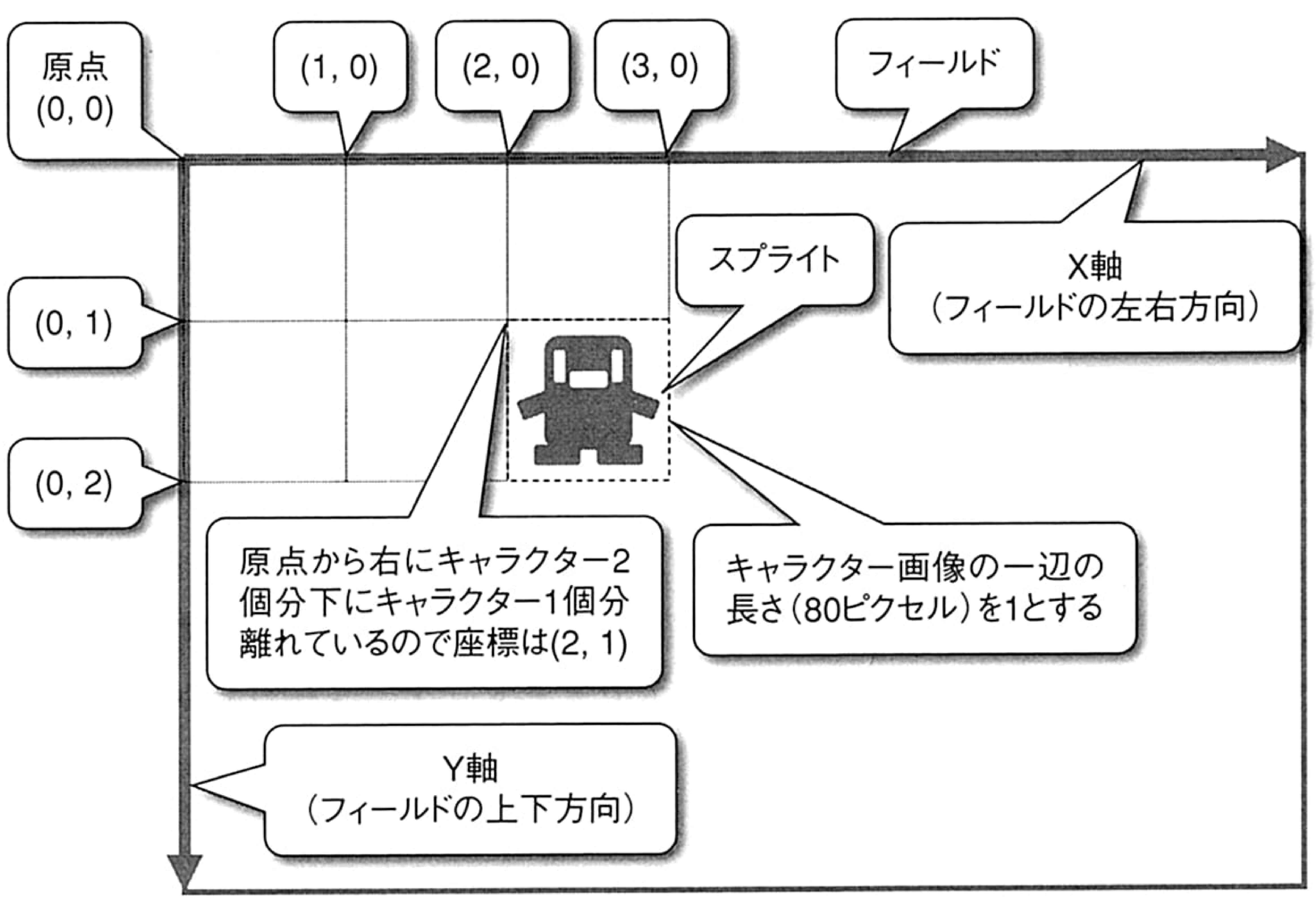


スプライトの座標

キャラクターの移動(p.39)では、キャラクターの座標を画面上のピクセル単位で指定しました。スプライトの座標も、同様にピクセル単位で指定してもよいのですが、ここでは少し違った方法を使うことにします。

本書ではキャラクターの画像を80×80ピクセルの正方形にしています。この画像の大きさ、つまり正方形の辺の長さである80ピクセルを1として、スプライトの座標を表すことにします(Fig.9)。

▼ Fig.9: スプライトの座標



このように座標を表すと、例えばキャラクターの座標が(x, y)のときに、座標を、

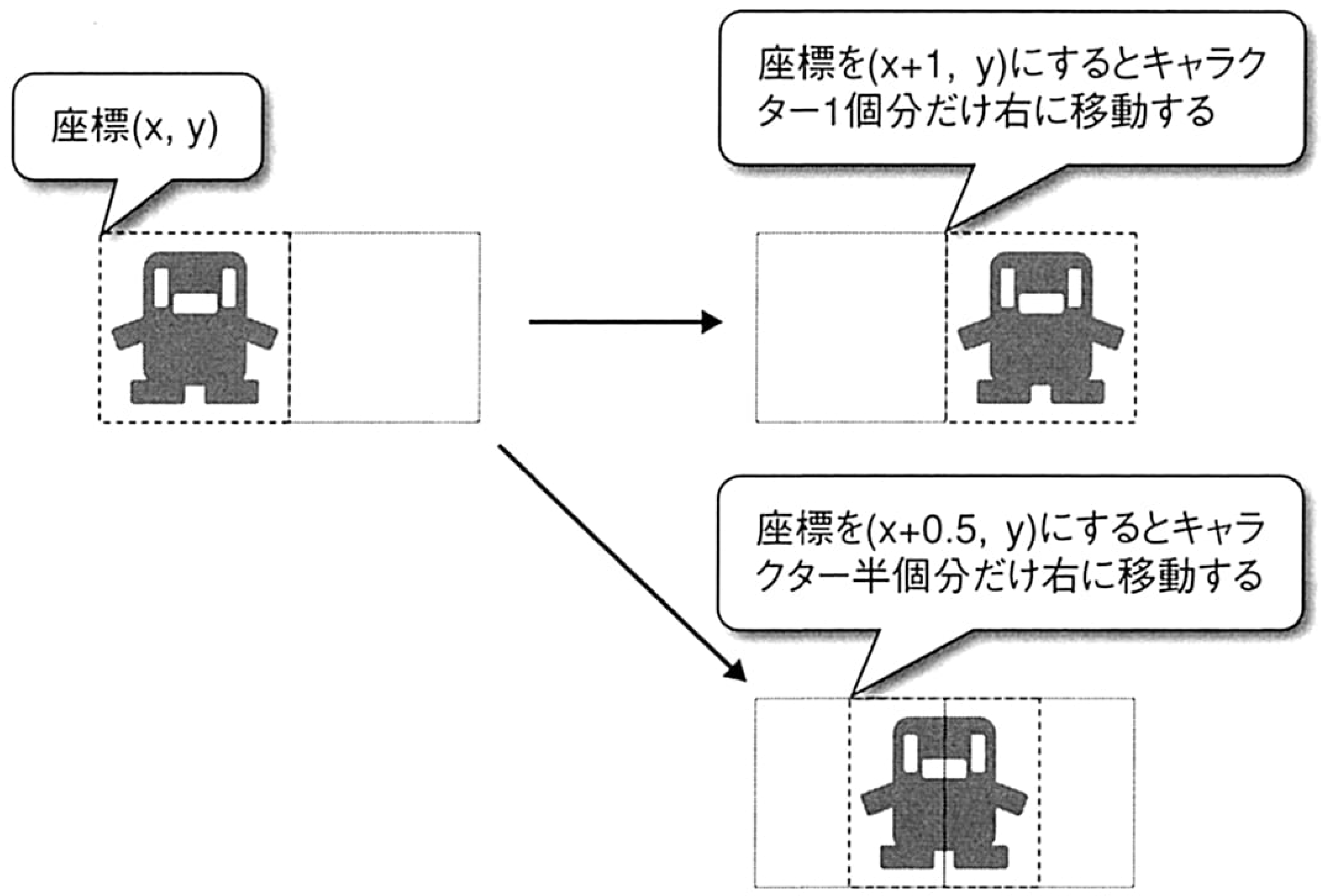
```
(x+1, y)
```

に変化させると、キャラクター1個分だけ右に移動させることができます。また、キャラクター半個分だけ右に移動させるときには、

```
(x+0.5, y)
```

とします(Fig.10)。

Fig.10: スプライトの移動



もしも座標がピクセル単位だと、キャラクター1個分だけ右に移動させるには、

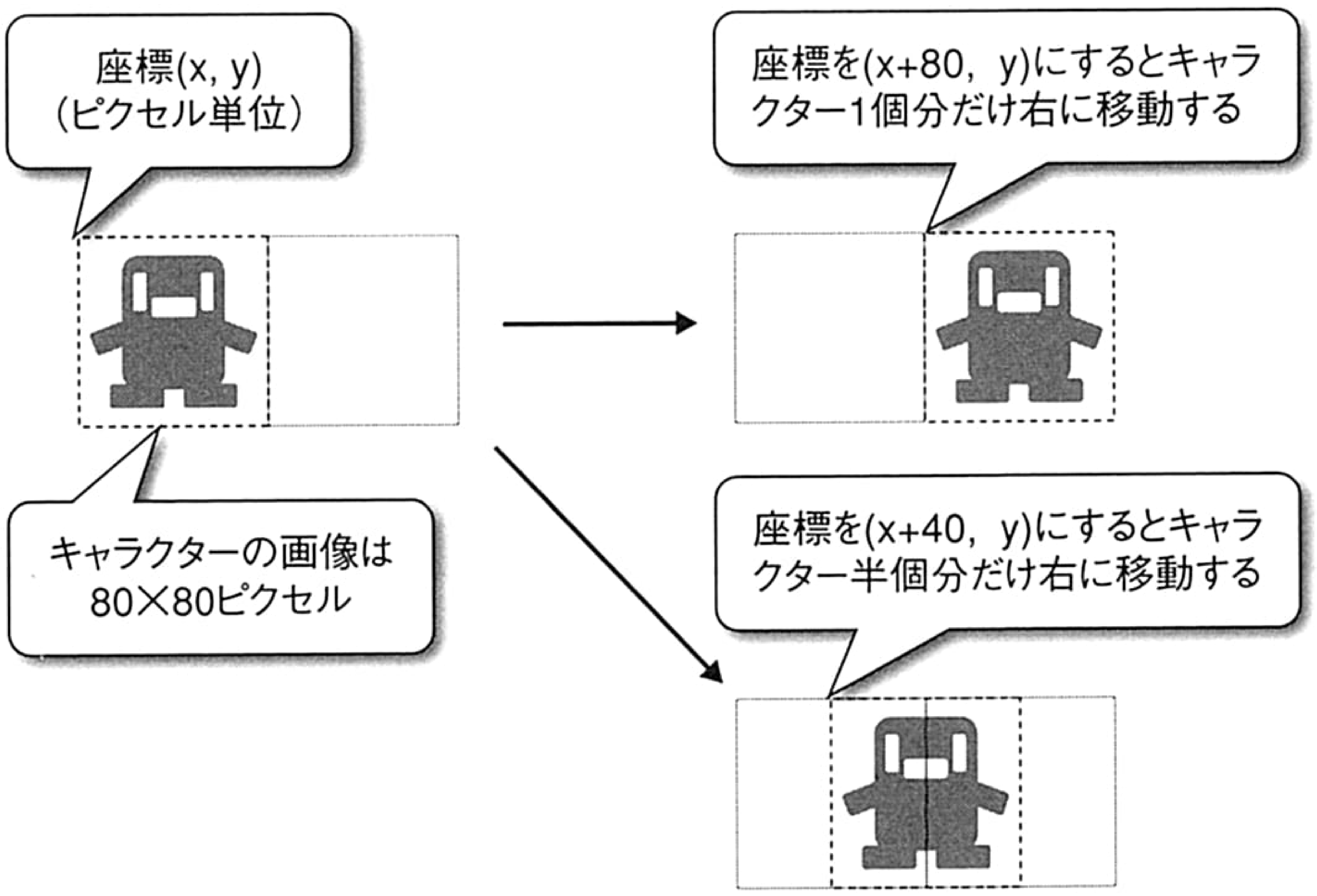
```
(x+80, y)
```

となり、半個分の場合には、

```
(x+40, y)
```

となります(Fig.11)。

▼ **Fig.11**: ピクセル単位でスプライトの座標を指定する



ゲームでは、キャラクター1個分や半個分といった移動をよく行うので、キャラクターの大きさを1とする座標にした方が、座標の計算が分かりやすくなります。また、画面の解像度を変えて、キャラクターの大きさが変わった場合にも、プログラムの変更が必要ないという利点もあります。

そこで本書では、キャラクターの大きさを1とする座標を使うことにします。

スプライトのプログラム

スプライトのプログラム(List2)は、ライブラリ(common.js)に含まれています。Gameオブジェクト(p.53のList1)と同様に、スプライトの機能はSpriteオブジェクトにまとめます。

List2では、

```
function Sprite(div, zIndex) {
```

のようにSpriteオブジェクト(Sprite関数)を定義します。divとzIndexは引数(関数に与えるパラメータ)です。

まず、スプライトの画像を表示するために、

```
obj.img=document.createElement("img");
div.appendChild(obj.img);
```

のように、createElement関数でimg要素を作成し、appendChild関数でdiv要素に登録します。

このimg要素については、

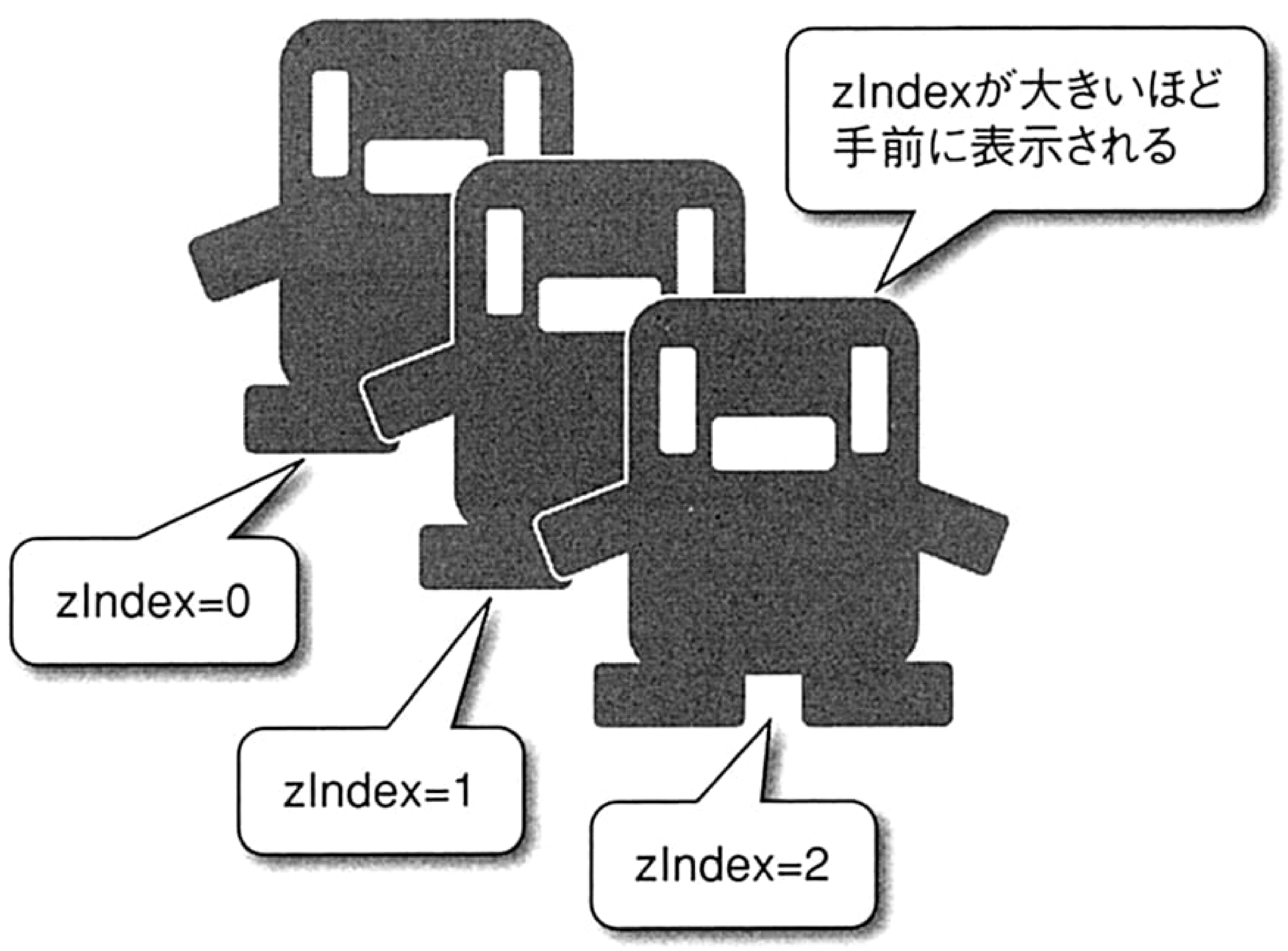
```
obj.img.src="void.png";
```

のようにsrcプロパティを設定します。void.pngは空白の画像です。また、

```
style=obj.img.style;
style.position="absolute";
style.zIndex=zIndex;
```

のように、表示位置の指定方法(絶対座標)と、画像を重ね合わせる順番を設定します。複数の画像を重ねて表示したときには、zIndexプロパティの値が大きい画像ほど、手前に表示されます(Fig.12)。例えば、背景の手前にキャラクターの画像を表示したいときなどに、zIndexプロパティを使います。

Fig.12: 画像の重ね合わせ



続いて、次のような関数を定義します。これらはSpriteオブジェクトに属する関数です。

- setXY関数　　：スプライトの座標を設定する
- getX関数、getY関数　　：スプライトの座標を取得する
- setSize関数　　：スプライトのサイズを設定する
- getXSize関数、getYSize関数　　：スプライトのサイズを取得する
- initAnim関数　　：アニメーションを初期化する
- stepAnim関数　　：アニメーションを進行させる
- isAnimOver関数　　：アニメーションが終了したかどうかを調べる

オブジェクトに属する関数を定義するには、

```
obj.関数名 = function(引数) {
    処理;
}
```

のような記法を使います。例えばsetXY関数は、

```
obj.setXY=function(x, y) {
    obj.updated=false;
    obj.x=x;
    obj.y=y;
}
```

のように定義しました。この関数では、updateプロパティにfalseを代入し、xプロパティとyプロパティには引数のxとyを代入しています。updatedプロパティは、表示を更新するかどうかを判断するために使います(p.68)。xプロパティとyプロパティはスプライトの座標です。

また、例えばgetX関数は、

```
obj.getX=function() {
    return obj.x;
}
```

のように定義しました。この関数はxプロパティの値、つまりスプライトのX座標を返します。return文は、関数の呼び出し元に値を返すための構文です。

Spriteオブジェクトに属する関数については、後ほど使用例を紹介します(p.74のList4)。

List2: スプライトのプログラム(common.js)

```
// スプライトのオブジェクト
// div:画像を登録するdiv要素
// zIndex:画像を重ね合わせる順番
function Sprite(div, zIndex) {

  // 変数objがオブジェクト自身を指すようにする
  var obj=this;

  // 関数内で一時的に使う変数の定義
  var style;
```

```
// img要素の作成
obj.img=document.createElement("img");
div.appendChild(obj.img);

// 空白の画像を表示しておく
obj.img.src="void.png";

// 表示位置の指定方法と、画像を重ね合わせる順番を設定する
style=obj.img.style;
style.position="absolute";
style.zIndex=zIndex;

// 位置
obj.setXY=function(x, y) {
  obj.updated=false;
  obj.x=x;
  obj.y=y;
}
obj.getX=function() {
  return obj.x;
}
obj.getY=function() {
  return obj.y;
}
obj.setXY(0, 0);

// サイズ
obj.setSize=function(xSize, ySize) {
  obj.updated=false;
  obj.xSize=xSize;
  obj.ySize=ySize;
}
obj.getXSize=function() {
  return obj.xSize;
}
obj.getYSize=function() {
  return obj.ySize;
}
obj.setSize(1, 1);

// アニメーション
obj.initAnim=function(array) {
  obj.updated=false;
  obj.animImage=array;
  obj.animIndex=0;
  obj.animOver=false;
}
```

```
  obj.stepAnim=function() {
    obj.updated=false;
    obj.animIndex++;
    if (obj.animIndex>=obj.animImage.length) {
      obj.animIndex=0;
      obj.animOver=true;
    }
  }
  obj.isAnimOver=function() {
    return obj.animOver;
  }
  obj.initAnim(new Array("void.png"));

  // ...(中略)...
}
```

スプライト

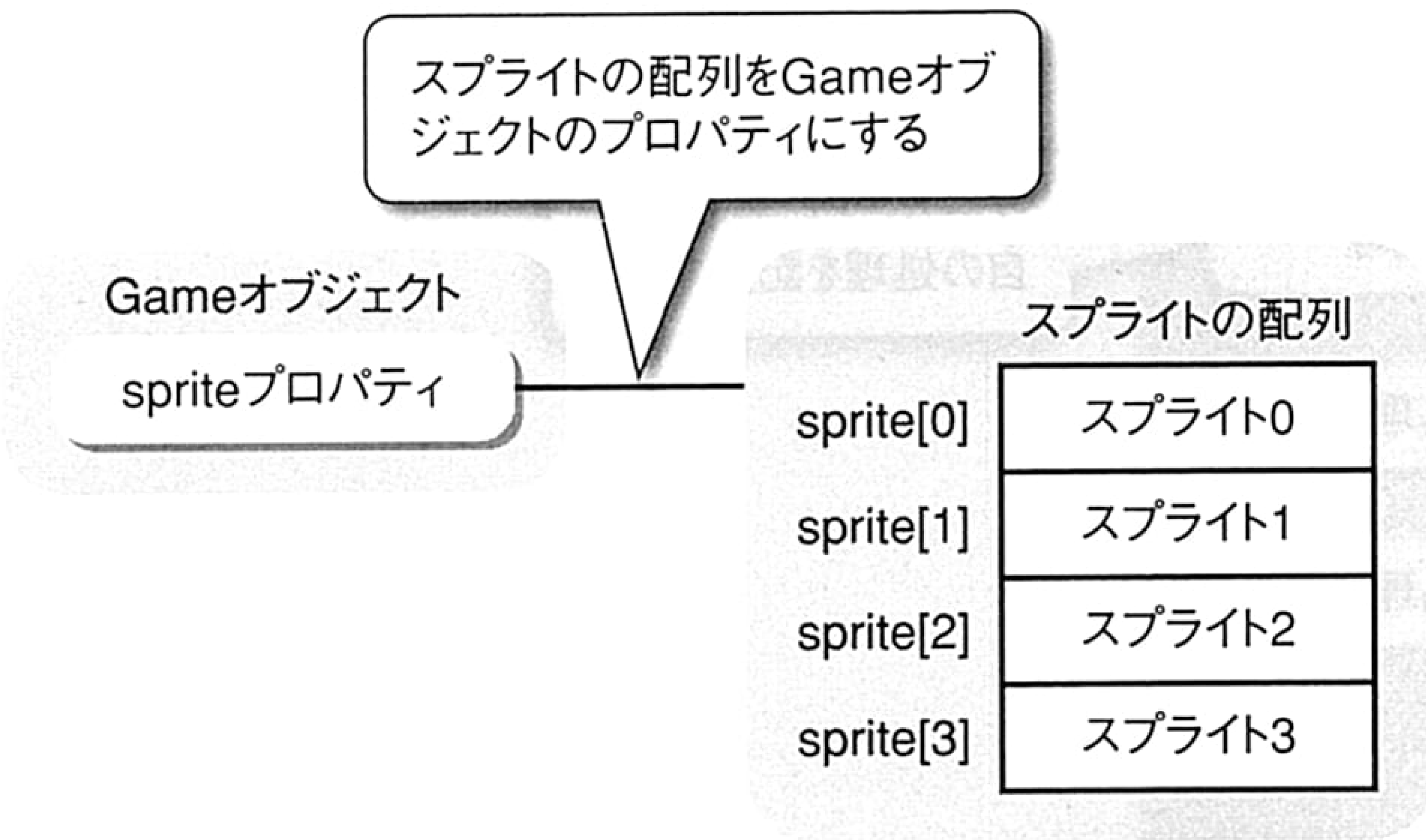
スプライトとは、ゲームの画面にキャラクターなどの画像を表示するための仕組みです。画像を移動させたり、複数の画像を重ね合わせたりといったことが簡単にできるので、主に2Dグラフィックスのゲームで利用されます。
昔のアーケードゲームやゲーム専用機などは、スプライト機能を提供する専用のハードウェアを搭載していました。現在ではプロセッサが高速になったため、ソフトウェアでスプライトと同等の機能を実現することも可能です。3Dグラフィックス用のハードウェアを使って、スプライトを実現する方法もあります。
本書で作成したようなスプライトは、元々のハードウェアによるスプライトとは別のものです。しかし、機能はよく似ているので、スプライトと呼んでいます。

表示の更新

キャラクターのアニメーション(p.29)や移動(p.39)で解説したように、キャラクターを動かすには、一定時間ごとに表示の更新を行う必要があります。ここでは、これまでに定義したゲーム本体のオブジェクト(Game)とスプライトのオブジェクト(Sprite)に対して、表示を更新する処理を組み込みます。

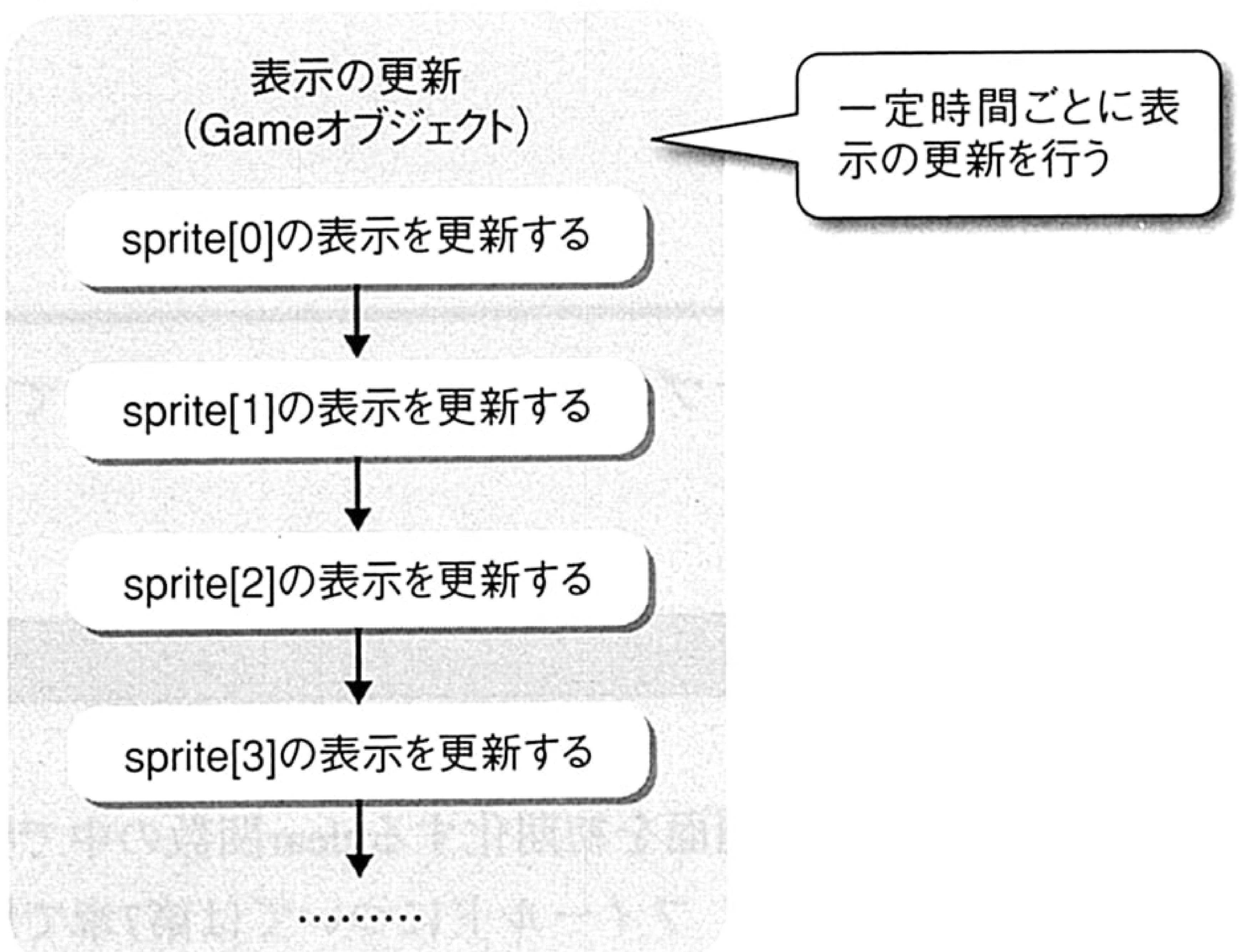
ゲームでは複数のスプライトを使います。そこで、これらのスプライトを配列で管理することにしましょう(Fig.13)。この配列はGameオブジェクトのspriteプロパティにします。

▼ Fig.13: スプライトを配列で管理する



Gameオブジェクトは、一定時間(例えば50ミリ秒など)ごとに表示の更新を行うことにします。スプライトの配列を使って、各Spriteオブジェクトの表示を更新します(Fig.14)。

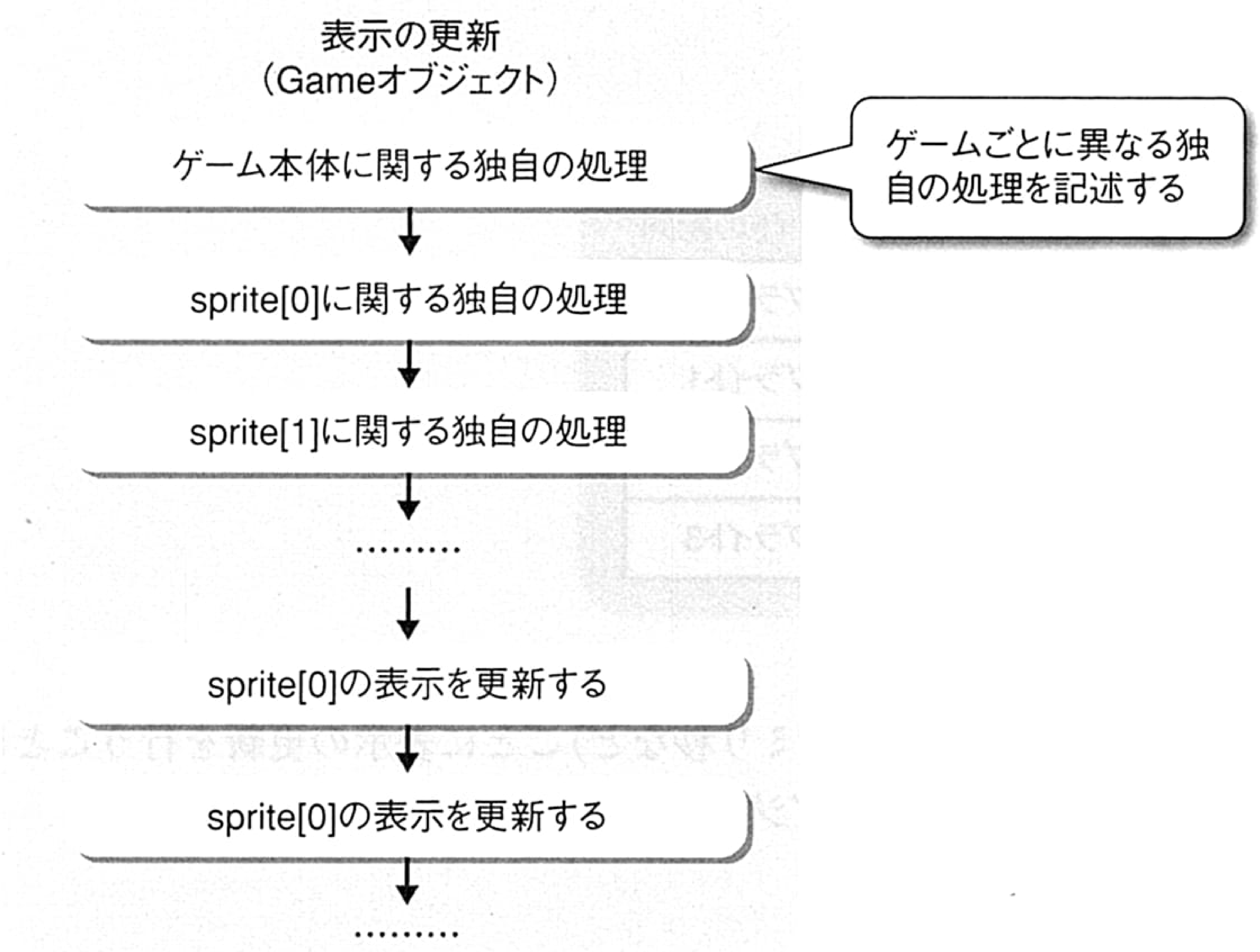
▼ Fig.14: スプライトの表示を更新する



これでスプライトの表示を更新することができますが、ゲームのライブラリとしては、これだけでは少し不十分です。例えばキャラクターの動きは、ゲームによって全く違います。ライブラリを利用しつつも、ゲームによって異なる動きをさせるには、一定時間ごとに行う処理をプログラマが独自に記述できる仕組みが必要です。

そこで、表示の更新を行う際に、ゲームごとに異なる独自の処理を呼び出すことにしましょう(Fig.15)。ゲーム本体とスプライトの各々について、独自の処理を記述できるようにします。

▼ **Fig.15**: ゲームごとに異なる処理を呼び出す



表示を更新するプログラム

表示を更新する処理は、GameオブジェクトとSpriteオブジェクトの両方にまたがっています(List3)。まずGameオブジェクトでは、

```
obj.sprite=new Array();
```

のように、スプライトの配列を初期化します。これは画面を初期化するclear関数の中で行います。なお、ここではフィールドの初期化も行いますが、フィールドについては第7章で解説します。

新しいスプライトを作るのはcreateSprite関数です。ここでは、

```
var s=new Sprite(obj.divField, obj.zIndex++);
```

として、Spriteオブジェクトを生成します。オブジェクトを生成するには、この例のように、

```
new オブジェクト名(引数)
```

と書きます。次の、

```
obj.sprite.push(s);
```

では、生成したSpriteオブジェクトを、配列spriteの末尾に追加します。pushは配列の末尾に要素を追加する関数です。

表示の更新はupdate関数で行います。Gameオブジェクトの最後に、

```
setInterval(obj.update, UPDATE_INTERVAL);
```

とあるように、一定時間(ここでは50ミリ秒)ごとにupdate関数を呼び出すように登録します。

update関数では、

```
obj.onUpdate();
```

のように、GameオブジェクトのonUpdate関数を呼び出します。onUpdate関数は、ゲームごとに異なる独自の処理を記述するための関数です。ここでは、

```
obj.onUpdate=function() {}
```

のように、関数の内容を空にしておきます。具体的な内容は、ゲームのプログラムで定義します。

次は、

```
for (i=0; i<obj.sprite.length; i++) {
    obj.sprite[i].onUpdate();
}
```

のようにfor文を使って、各スプライトのonUpdate関数を呼び出します。スプライトのonUpdate関数にも、ゲームごとに異なる独自の処理を記述することができます。

続いて、

```
for (i=0; i<obj.sprite.length; i++) {
    obj.sprite[i].update(obj.viewX, obj.viewY);
}
```

では、各スプライトのupdate関数を呼び出して、スプライトの表示を更新します。スプライトのupdate関数では、

```
var style=obj.img.style;
style.left=Math.floor((obj.x-viewX)*CELL_SIZE)+"px";
style.top=Math.floor((obj.y-viewY)*CELL_SIZE)+"px";
style.width=Math.floor(obj.xSize*CELL_SIZE)+"px";
style.height=Math.floor(obj.ySize*CELL_SIZE)+"px";
```

のように座標とサイズを設定します。前述のように座標とサイズの単位は、キャラクターの大きさを1としています。また、

```
obj.img.src=obj.animImage[obj.animIndex];
```

では、スプライトの画像を入れ替えることによって、アニメーションを再生します。

スプライトのupdate関数では、updatedプロパティを使って、表示の更新を行うかどうかを制御しています。座標やサイズなどが変化した場合に限って、表示を更新することにより、不要な処理を省いています。

また、変数viewX, viewYはフィールドのスクロールに使います。スクロールについては第7章で解説します。

List3: 表示を更新するプログラム(common.js)

```
// ゲーム本体
function Game() {
  // ...(中略)...

  // 画面のクリア
  obj.clear=function() {

    // フィールドを空にする
    obj.divField.innerHTML="";

    // スプライトとフィールドの配列を作る
    obj.sprite=new Array();
    obj.field=new Array();

    // 重ね合わせの順序を初期化する
    obj.zIndex=0;
  }

  // 画面をクリアする
  obj.clear();
```

```
  // スプライトの作成
  obj.createSprite=function() {
    var s=new Sprite(obj.divField, obj.zIndex++);
    obj.sprite.push(s);
    return s;
  }

  // 更新時に行いたい独自の処理を記述する関数
  obj.onUpdate=function() {}

  // 表示の更新
  obj.update=function() {

    // 更新時の処理を呼び出す
    obj.onUpdate();
    var i;
    for (i=0; i<obj.field.length; i++) {
      obj.field[i].onUpdate();
    }
    for (i=0; i<obj.sprite.length; i++) {
      obj.sprite[i].onUpdate();
    }

    // 表示を更新する
    for (i=0; i<obj.field.length; i++) {
      obj.field[i].update(obj.viewX, obj.viewY);
    }
    for (i=0; i<obj.sprite.length; i++) {
      obj.sprite[i].update(obj.viewX, obj.viewY);
    }
  }

  // 一定時間ごとに表示の更新(update関数)を実行する
  setInterval(obj.update, UPDATE_INTERVAL);
}

// スプライト
function Sprite(div, zIndex) {
  // ...(中略)...

  // 更新済みかどうか
  obj.updated=false;

  // 視点の座標
  obj.viewX=0;
  obj.viewY=0;
```

```
  // 更新時に行いたい独自の処理を記述する関数
  obj.onUpdate=function() {}

  // 表示の更新
  obj.update=function(viewX, viewY) {

    // 表示が更新済みで、視点の座標も変化していなければ、更新は行わない
    if (obj.updated && viewX==obj.viewX && viewY==obj.viewY) return;

    // 更新済みにする
    obj.updated=true;

    // 視点の座標を記録する
    obj.viewX=viewX;
    obj.viewY=viewY;

    // 画像の座標とサイズを更新する
    var style=obj.img.style;
    style.left=Math.floor((obj.x-viewX)*CELL_SIZE)+"px";
    style.top=Math.floor((obj.y-viewY)*CELL_SIZE)+"px";
    style.width=Math.floor(obj.xSize*CELL_SIZE)+"px";
    style.height=Math.floor(obj.ySize*CELL_SIZE)+"px";

    // アニメーションのために画像を入れ替える
    obj.img.src=obj.animImage[obj.animIndex];
  }
}
```

for文

for文は繰り返しを行うための構文です。記法は、

```
for (式1; 式2; 式3) {
    処理;
}
```

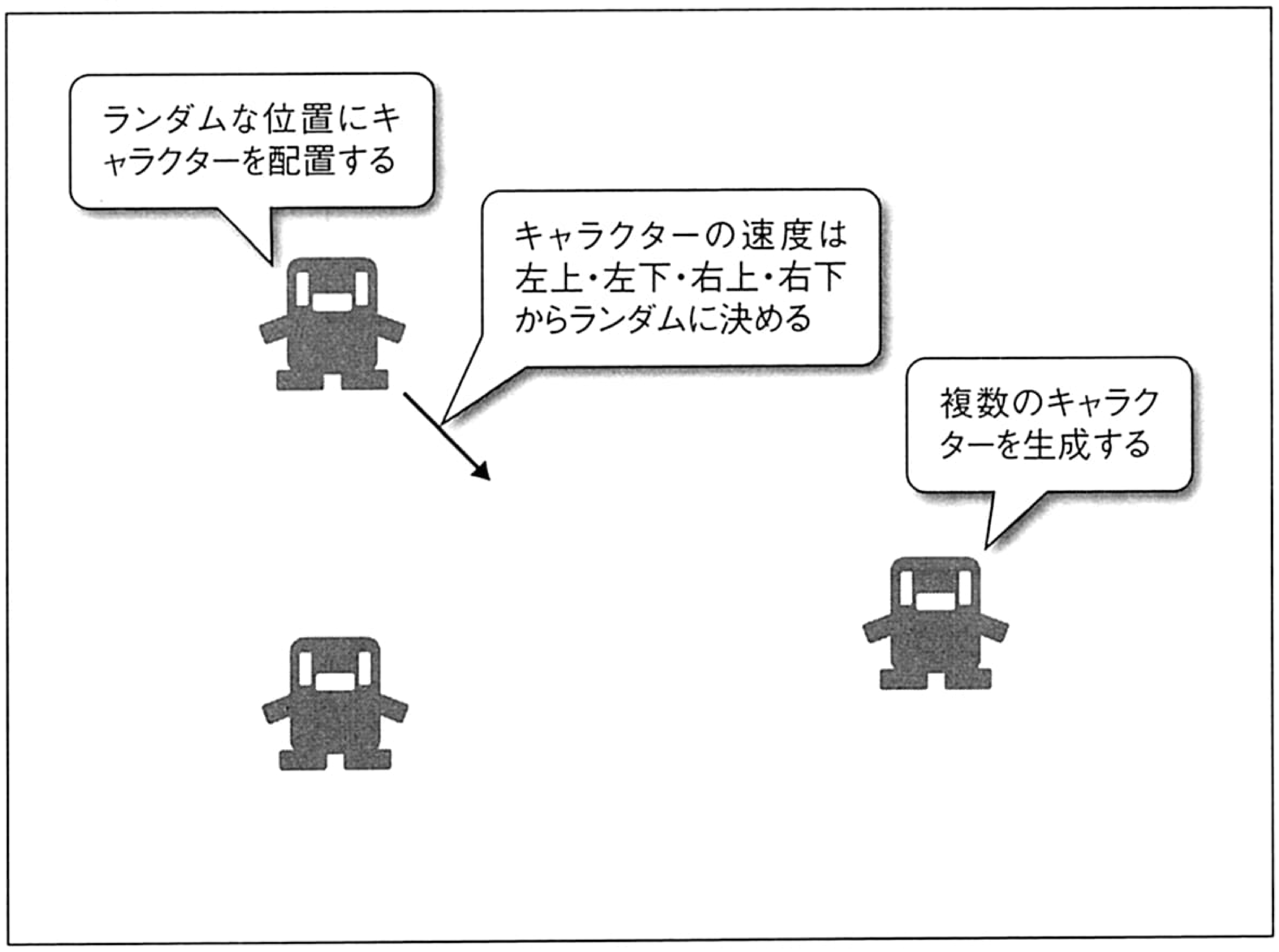
です。最初に式1を実行し、次に処理と式3を実行します。あとは式2がtrueである限り、処理と式3の実行を繰り返します。

画面上を動き回るキャラクター群

ライブラリが整ってきたところで、ライブラリを使ったプログラムを作ってみましょう。ここではキャラクターの移動(p.39)を発展させて、多数のキャラクターが画面上を動き回るプログラムを作ることにします。

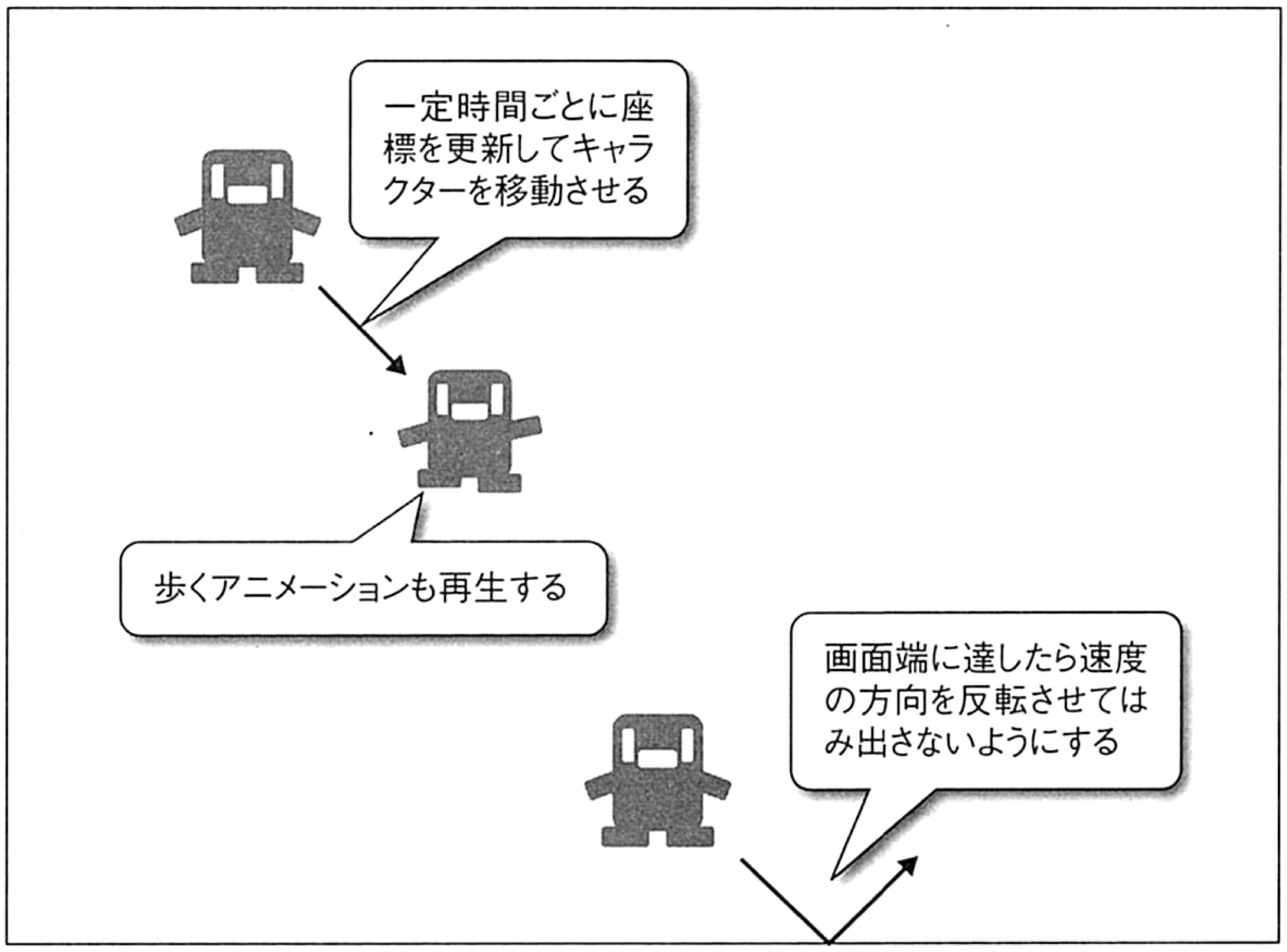
画面上のランダムな位置にキャラクターを配置します(Fig.16)。キャラクターの速度もランダムに決めて、左上･左下･右上･右下のいずれかの方向に進むようにします。

Fig.16: キャラクターの配置



一定時間ごとにキャラクターの座標を更新することによって、キャラクターを移動させます(Fig.17)。このとき、キャラクターが歩くアニメーションも再生します。キャラクターが画面端に達したら、速度の方向を反転させて、画面からはみ出さないようにします。

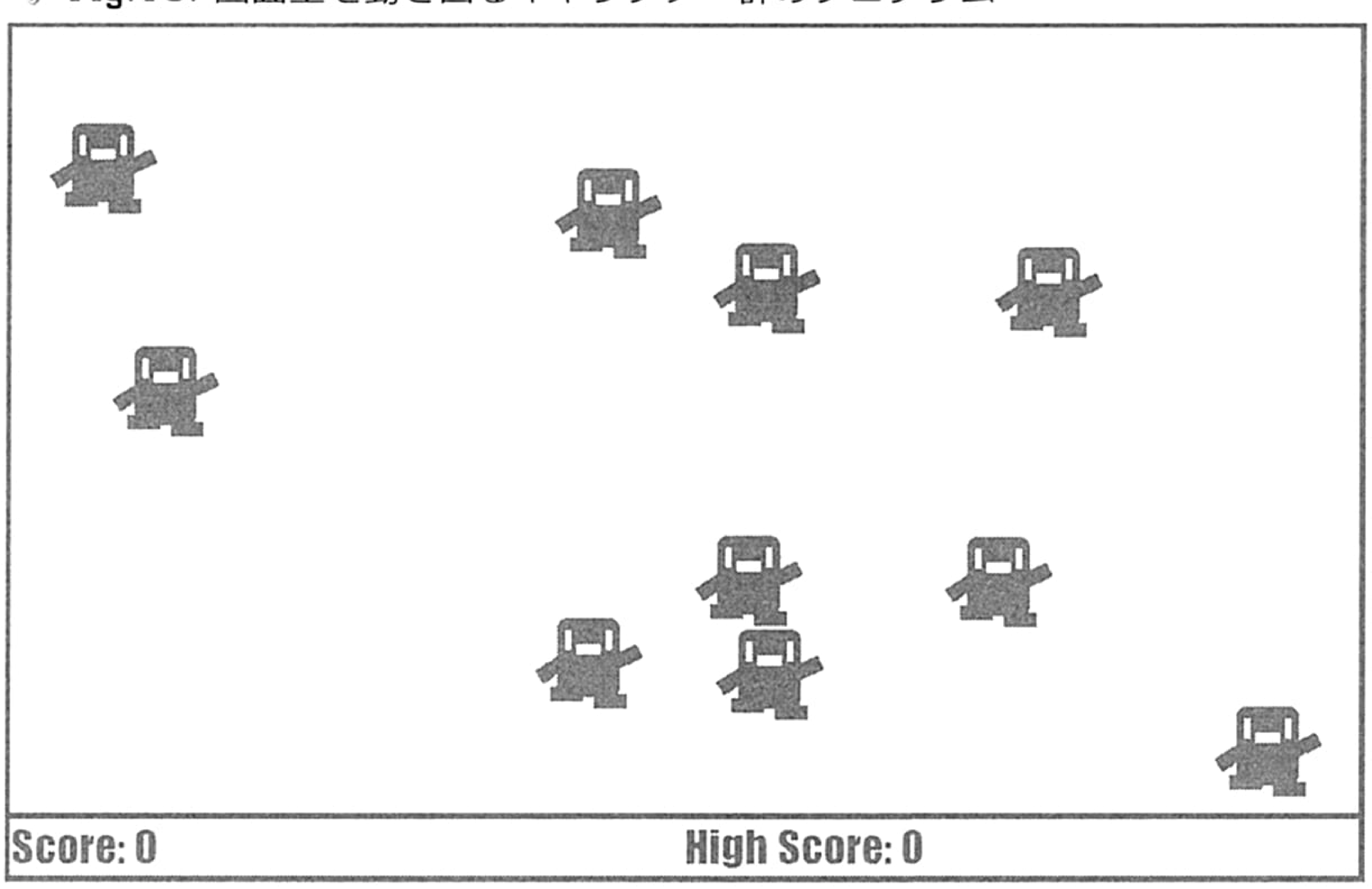
▼ **Fig.17**: キャラクターの移動



画面上を動き回るキャラクター群のプログラム

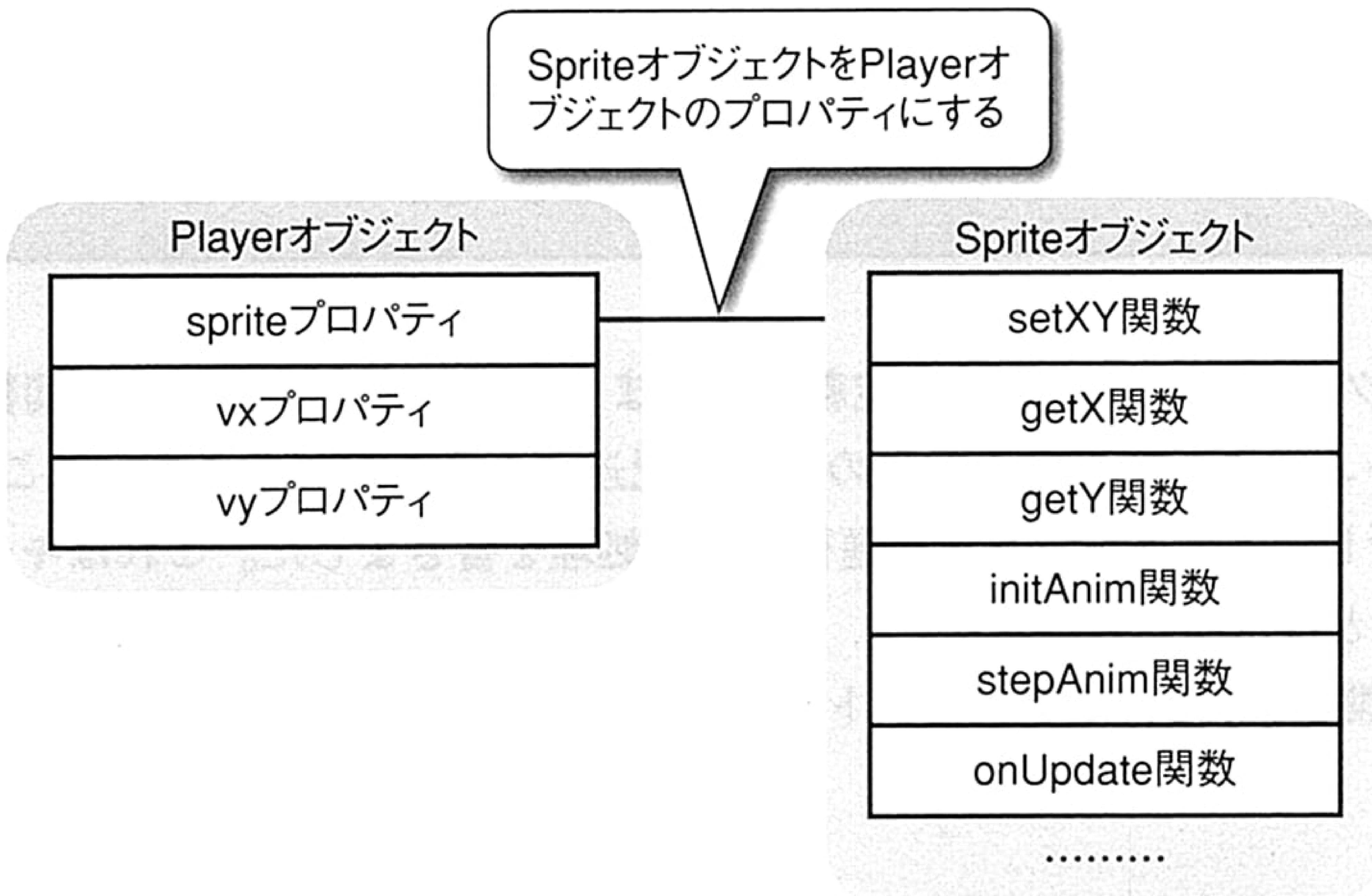
List4は画面上を動き回るキャラクター群のプログラムです。HTMLファイル(walk.html)をブラウザで開くと、プログラムを実行することができます(Fig.18)。10体のキャラクターが出現し、画面内を歩き回ります。

▼ **Fig.18**: 画面上を動き回るキャラクター群のプログラム



List4では、キャラクターに関する処理をPlayerオブジェクトにまとめています(Fig.19)。Playerオブジェクトは、前述のSpriteオブジェクトをプロパティに持っています。また、速度を表すvx, vyもプロパティに持ちます。

Fig.19: Playerオブジェクト



スプライトは、

```
obj.sprite=game.createSprite();
```

のように、createSprite関数を使って作成します。次に、

```
obj.sprite.initAnim(playerWalkAnim);
```

のようにアニメーションを初期化し、

```
obj.sprite.setXY(randFloat(XMAX), randFloat(YMAX));
obj.vx=randSign()*PLAYER_SPEED;
obj.vy=randSign()*PLAYER_SPEED;
```

のように座標や速度をランダムに決めます。

アニメーションの内容については、main関数の中で、

```
playerWalkAnim=new Array(
    "pcWalk0.png", "pcWalk2.png", "pcWalk0.png",
```

```
    "pcWalk1.png", "pcWalk3.png", "pcWalk1.png"
);
```

のように初期化しています。これはアニメーションの表示(p.36のList4)と同じく、アニメーションに使う画像ファイル名の一覧です。

Playerオブジェクトでは、

```
obj.sprite.onUpdate=function() {
```

として、SpriteオブジェクトのonUpdate関数を定義しています。このようにonUpdate関数を定義することにより、ゲームによって異なる独自の処理を記述することができます。ここでは、スプライトの座標を更新し、アニメーションを進行させる処理を書きました。これはキャラクターの移動(p.45のList5)とほぼ同じ処理です。

メインルーチン(main関数)では、オブジェクトや配列の初期化を行います。Gameオブジェクトについては、

```
game=new Game();
```

のように作成します。キャラクターについては、

```
player=new Array();
```

のように配列を初期化しておき、

```
for (i=0; i<PLAYER_COUNT; i++) {
    player[i]=new Player();
}
```

のようにPlayerオブジェクトを作成します。

List4: 画面上を動き回るキャラクター群のプログラム(walk.js)

```
// キャラクターの数
var PLAYER_COUNT=10;

// キャラクターの移動スピード
var PLAYER_SPEED=0.05;
```

```
// ゲーム本体のオブジェクト
var game;

// キャラクターの配列
var player;

// アニメーションの配列
var playerWalkAnim;

// キャラクターのオブジェクト
function Player() {

  // 変数objがオブジェクト自身を指すようにする
  var obj=this;

  // スプライトの作成
  obj.sprite=game.createSprite();

  // アニメーションの初期化
  obj.sprite.initAnim(playerWalkAnim);

  // 座標をランダムに決める
  obj.sprite.setXY(randFloat(XMAX), randFloat(YMAX));

  // 速度をランダムに決める
  obj.vx=randSign()*PLAYER_SPEED;
  obj.vy=randSign()*PLAYER_SPEED;

  // 一定時間ごとに呼び出す更新処理
  obj.sprite.onUpdate=function() {

    // 移動後の座標を求める
    var x=obj.sprite.getX()+obj.vx;
    var y=obj.sprite.getY()+obj.vy;

    // 画面外に出るならば、速度の方向を逆にする
    if (x<0 || x>XMAX) obj.vx=-obj.vx;
    if (y<0 || y>YMAX) obj.vy=-obj.vy;

    // スプライトの座標を更新する
    obj.sprite.setXY(x, y);

    // アニメーションを進行させる
    obj.sprite.stepAnim();
  }
}

// メインルーチン
```

```
function main() {

  // 関数内で一時的に使う変数の定義
  var i;

  // ゲーム本体のオブジェクトを初期化する
  game=new Game();

  // キャラクターの配列を初期化する
  player=new Array();

  // アニメーションの配列を初期化する
  playerWalkAnim=new Array(
    "pcWalk0.png", "pcWalk2.png", "pcWalk0.png",
    "pcWalk1.png", "pcWalk3.png", "pcWalk1.png"
  );

  // キャラクターのオブジェクトを作成する
  for (i=0; i<PLAYER_COUNT; i++) {
    player[i]=new Player();
  }
}
```

乱数の生成

List4では、乱数を生成するためにいくつかの関数を使っています。乱数の生成は色々なゲームで使うので、これらの関数はライブラリ(common.js)で定義することにしました(List5)。

JavaScriptで乱数を生成するには、

```
Math.random()
```

のように、Mathオブジェクトのrandom関数を使います。random関数は0以上1未満の実数をランダムに生成します。

List5のrandFloat2関数は、random関数を使って、指定した範囲のランダムな実数を生成します。List5の他の関数は、このrandFloat2関数を利用して定義しました。

ランダムな整数を生成するrandInt2関数では、実数を整数に変換するために、Mathオブジェクトのfloor関数を使います。floor関数は、与えられた実数を超えない最大の整数を返します。例えば、

```
Math.floor(2.1)
```

は2になり、

```
Math.floor(-3.2)
```

は－4になります。

▼ **List5**: 乱数を生成するプログラム(common.js)

```
// from以上to未満のランダムな実数を生成する
function randFloat2(from, to) {
  return Math.random()*(to-from)+from;
}

// 0以上to未満のランダムな実数を生成する
function randFloat(to) {
  return randFloat2(0, to);
}

// from以上to未満のランダムな整数を生成する
function randInt2(from, to) {
  return Math.floor(randFloat2(from, to));
}

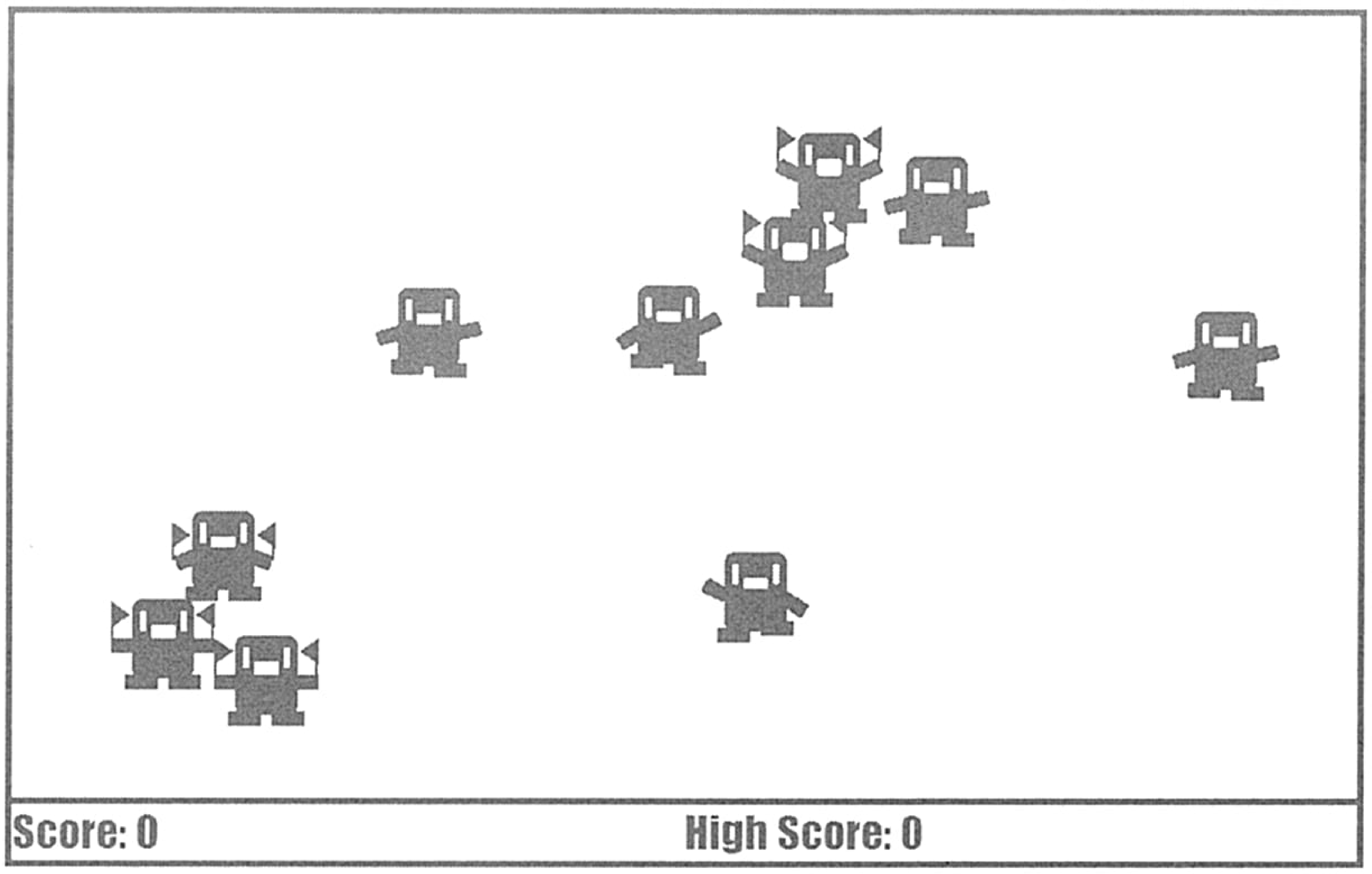
// 0以上to未満のランダムな整数を生成する
function randInt(to) {
  return randInt2(0, to);
}

// -1または1をランダムに生成する
function randSign() {
  return randInt(2)*2-1;
}
```

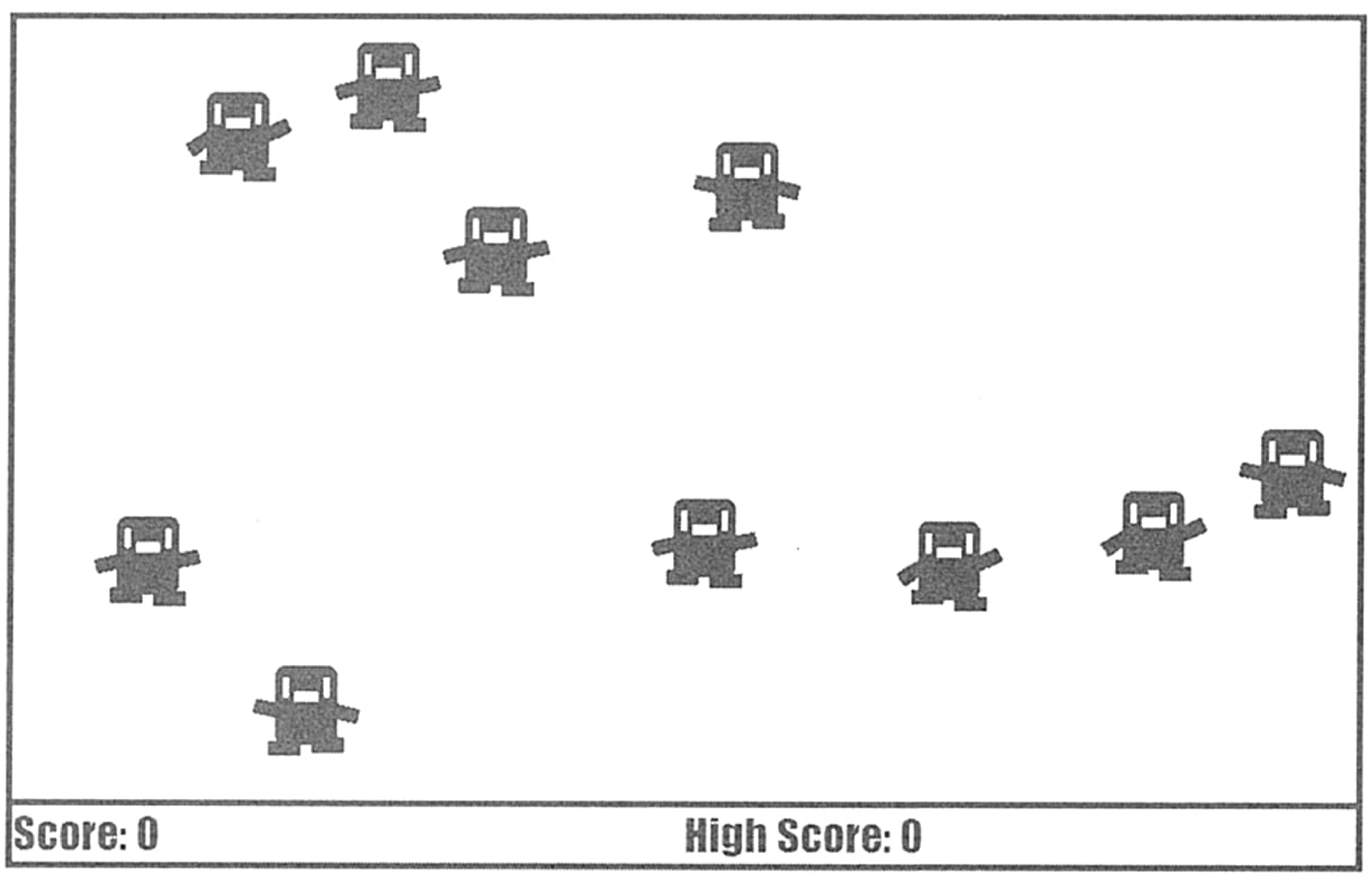
出会うと挨拶をするキャラクター群

画面上を歩き回るキャラクター群のプログラムを応用して、キャラクター同士が出会ったときに挨拶するプログラムを作ってみましょう。歩き回っているキャラクター同士が接触すると、両手で旗を振って挨拶します(Fig.20)。距離が離れると、キャラクターは挨拶をやめます(Fig.21)。

▼ Fig.20: 出会うと挨拶をする

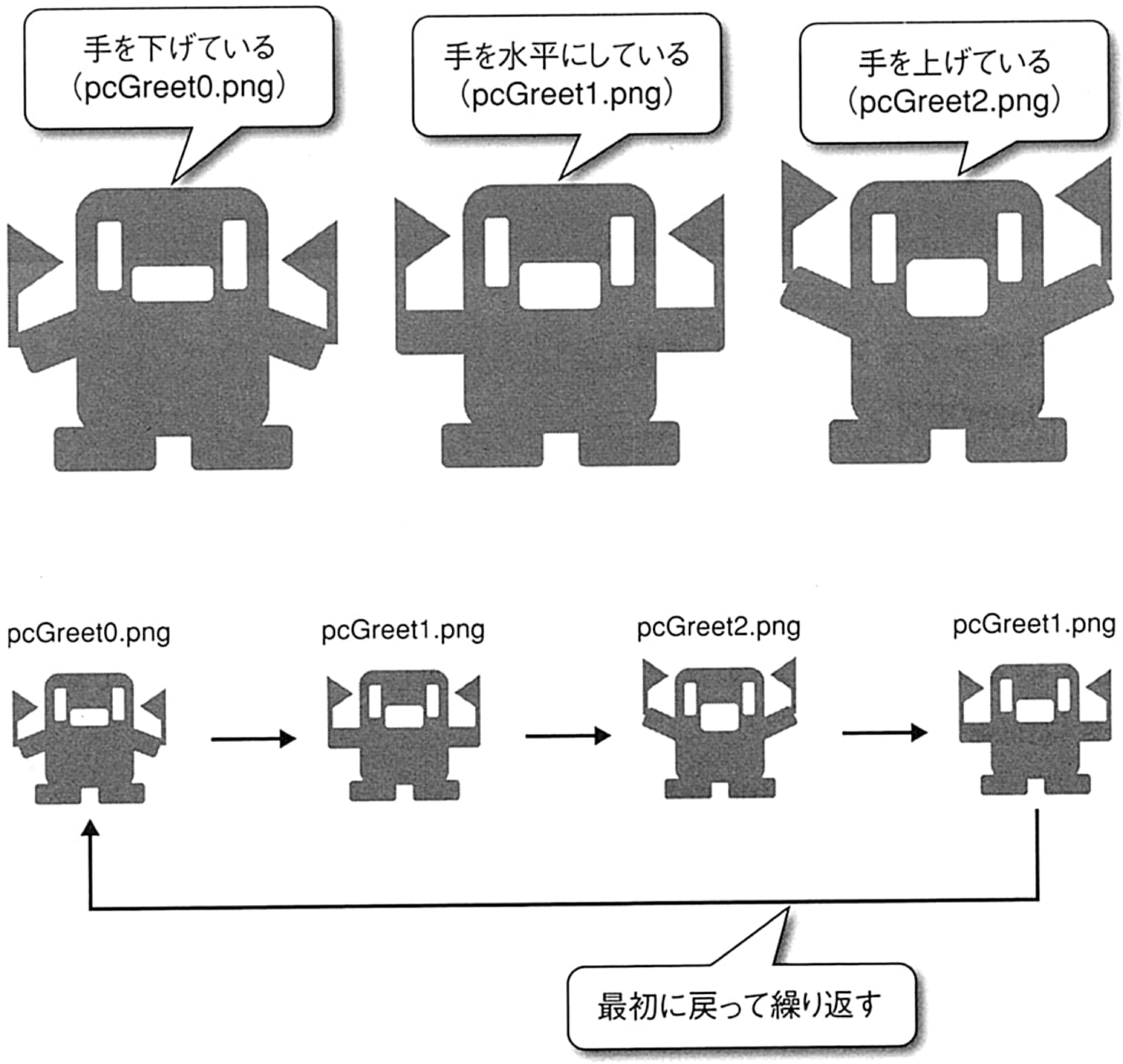


▼ Fig.21: 離れると挨拶をやめる



挨拶をするアニメーションのために、3枚の画像を用意しました(Fig.22)。この画像を並べて、4コマのアニメーションを作ります。

▼ **Fig.22**: 挨拶をするアニメーション



キャラクター同士が接触したら、歩くアニメーションから挨拶するアニメーションに切り替えます。キャラクター間の距離が離れたら、再び歩くアニメーションに戻します。

このプログラムの課題は、キャラクター同士が接触したかどうかを判定する方法です。それには当たり判定処理を使います。当たり判定処理とは、主にゲームプログラムにおいて、物体同士が接触したかどうかを判断する処理のことです。

本書では、当たり判定処理の具体的な方法を第5章で解説します。ライブラリ（common.js）には、スプライト同士の当たり判定処理を行うための関数（testHitSprite関数）を用意してあるので、ここではその関数を使うことにします。

挨拶をするキャラクターのプログラム

List6は挨拶をするキャラクターのプログラムです。画面上を動き回るキャラクター群のプログラム（p.74のList4）に対して、挨拶に関する処理を追加しました。

メインルーチン（main関数）では、

```
playerGreetAnim=new Array(
    "pcGreet0.png", "pcGreet1.png",
    "pcGreet2.png", "pcGreet1.png"
);
```

のように、挨拶するアニメーションを定義します。

キャラクターを表すPlayerオブジェクトには、

```
obj.greet=false;
```

のように、挨拶をしているかどうかを表すプロパティを追加します。挨拶をしているときはtrue、していないときはfalseです。

一定時間ごとに呼び出す更新処理（onUpdate関数）には、

```
if (game.testHitSprite(obj.sprite)) {
    if (!obj.greet) {
        obj.sprite.initAnim(playerGreetAnim);
        obj.greet=true;
    }
} else {
    if (obj.greet) {
        obj.sprite.initAnim(playerWalkAnim);
        obj.greet=false;
    }
}
```

のような処理を追加しました。testHitSprite関数は、指定したスプライトが他のスプライトに1つでも接触したときに、trueを返します。ここでは、

- 他のスプライトに接触しており、かつ挨拶をしていないときには、挨拶するアニメーションに切り替える
- 他のスプライトに接触しておらず、かつ挨拶をしているときには、歩くアニメーションに切り替える

という処理を行います。これでキャラクターは、他のキャラクターに接触している間だけ挨拶をするようになります。

▼ **List6**: 挨拶をするキャラクターのプログラム(greet.js)

```
// 定数
var PLAYER_COUNT=10;
var PLAYER_SPEED=0.05;

// ゲーム本体、キャラクター
var game;
var player;

// アニメーション
var playerWalkAnim;
var playerGreetAnim;

// キャラクターのオブジェクト
function Player() {
  var obj=this;

  // スプライトの初期化
  obj.sprite=game.createSprite();
  obj.sprite.initAnim(playerWalkAnim);
  obj.sprite.setXY(randFloat(XMAX), randFloat(YMAX));

  // 速度の初期化
  obj.vx=randSign()*PLAYER_SPEED;
  obj.vy=randSign()*PLAYER_SPEED;

  // 挨拶しているかどうかを表すプロパティ
  obj.greet=false;

  // 一定時間ごとに呼び出す更新処理
  obj.sprite.onUpdate=function() {

    // 座標の更新
    var x=obj.sprite.getX()+obj.vx;
    var y=obj.sprite.getY()+obj.vy;
    if (x<0 || x>XMAX) obj.vx*=-1;
    if (y<0 || y>YMAX) obj.vy*=-1;
    obj.sprite.setXY(x, y);

    // アニメーションの進行
    obj.sprite.stepAnim();

    // 他のキャラクターに接触したとき
    if (game.testHitSprite(obj.sprite)) {

      // 挨拶をしていなかったら、挨拶を開始する
      if (!obj.greet) {
```

```
        obj.sprite.initAnim(playerGreetAnim);
        obj.greet=true;
      }
    } else

    // 他のキャラクターに接触していないとき
    {
      // 挨拶をしていたら、挨拶を終了する
      if (obj.greet) {
        obj.sprite.initAnim(playerWalkAnim);
        obj.greet=false;
      }
    }
  }
}

// メインルーチン
function main() {
  var i;

  // ゲームのオブジェクトを初期化する
  game=new Game();

  // キャラクターの配列を初期化する
  player=new Array();

  // 歩くアニメーション
  playerWalkAnim=new Array(
    "pcWalk0.png", "pcWalk2.png", "pcWalk0.png",
    "pcWalk1.png", "pcWalk3.png", "pcWalk1.png"
  );

  // 挨拶するアニメーション
  playerGreetAnim=new Array(
    "pcGreet0.png", "pcGreet1.png",
    "pcGreet2.png", "pcGreet1.png"
  );

  // キャラクターのオブジェクトを初期化する
  for (i=0; i<PLAYER_COUNT; i++) {
    player[i]=new Player();
  }
}
```

Chapter 3

自機

自機を操作する

多くのゲームでは、プレイヤーが自分でキャラクターを操作することができます。そこで本書では、自分で動かすキャラクターを自機(じき)と呼ぶことにしましょう。これは主にシューティングゲームでプレイヤーが操作する飛行機や宇宙船などを指しますが、他のジャンルのゲームでも自機という言葉を使うことがあります。

自機を操作するための入力装置はゲームによって色々です。主に、

- キーボード
- マウス
- トラックボール
- ジョイスティック
- ジョイパッド
- ハンドル
- ガン(銃)
- リモコン

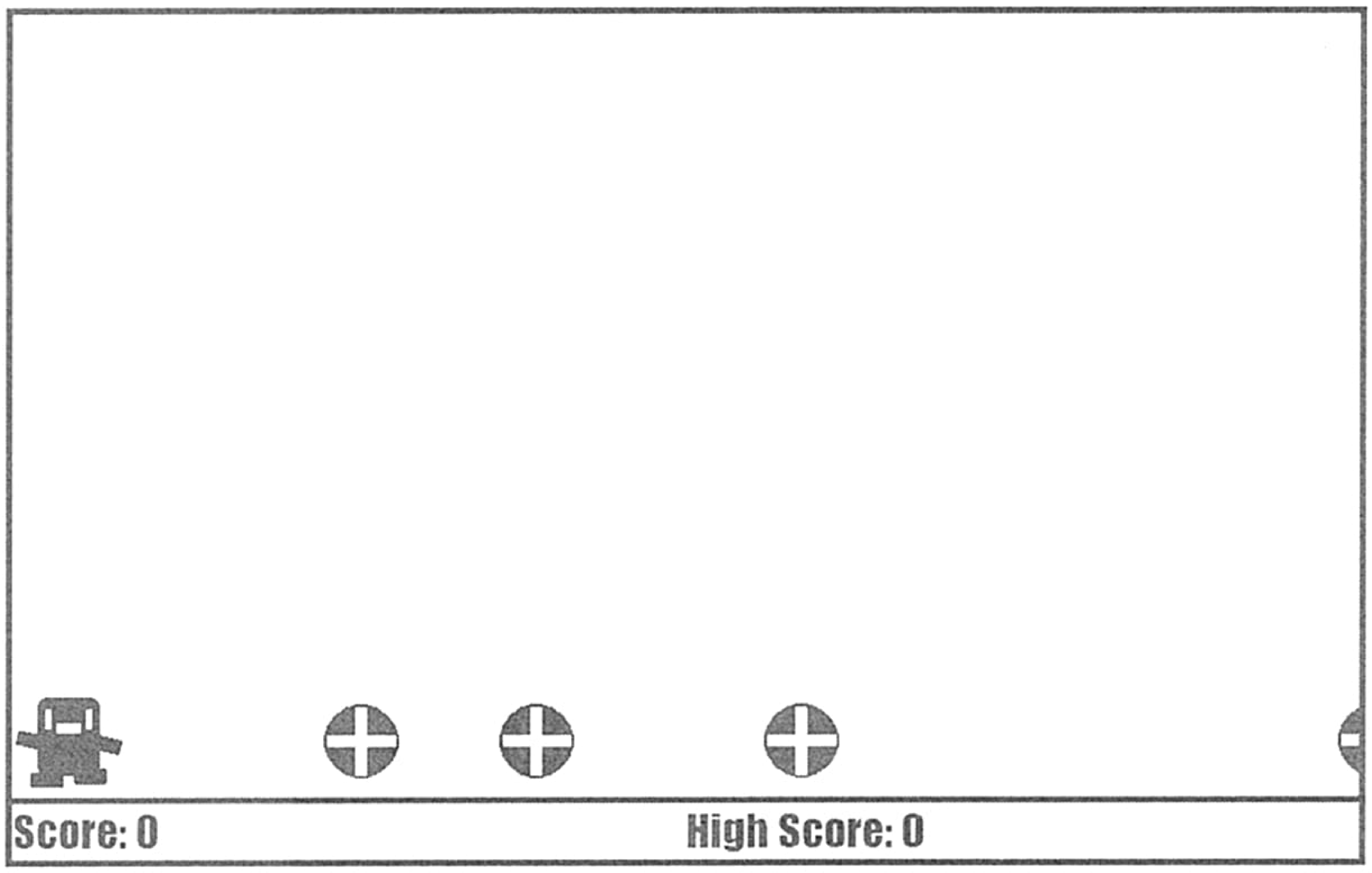
などが使われます。少し変わったものとしては、釣りゲームに使う釣り竿状のコントローラなどもあります。

ブラウザで遊ぶJavaScriptゲームやFlashゲームでは、入力装置としてキーボードやマウスを使うことが一般的です。本書ではキーボードとマウスを使って自機を操作する方法を紹介します。

自機を操作する例として、ゲーム仕立てのサンプルを作りました。キーボードで自機を移動させたりジャンプさせたりして、向かってくる障害物をかわすゲームです。実行するには、ブラウザでjumpKey.htmlを開いてください。

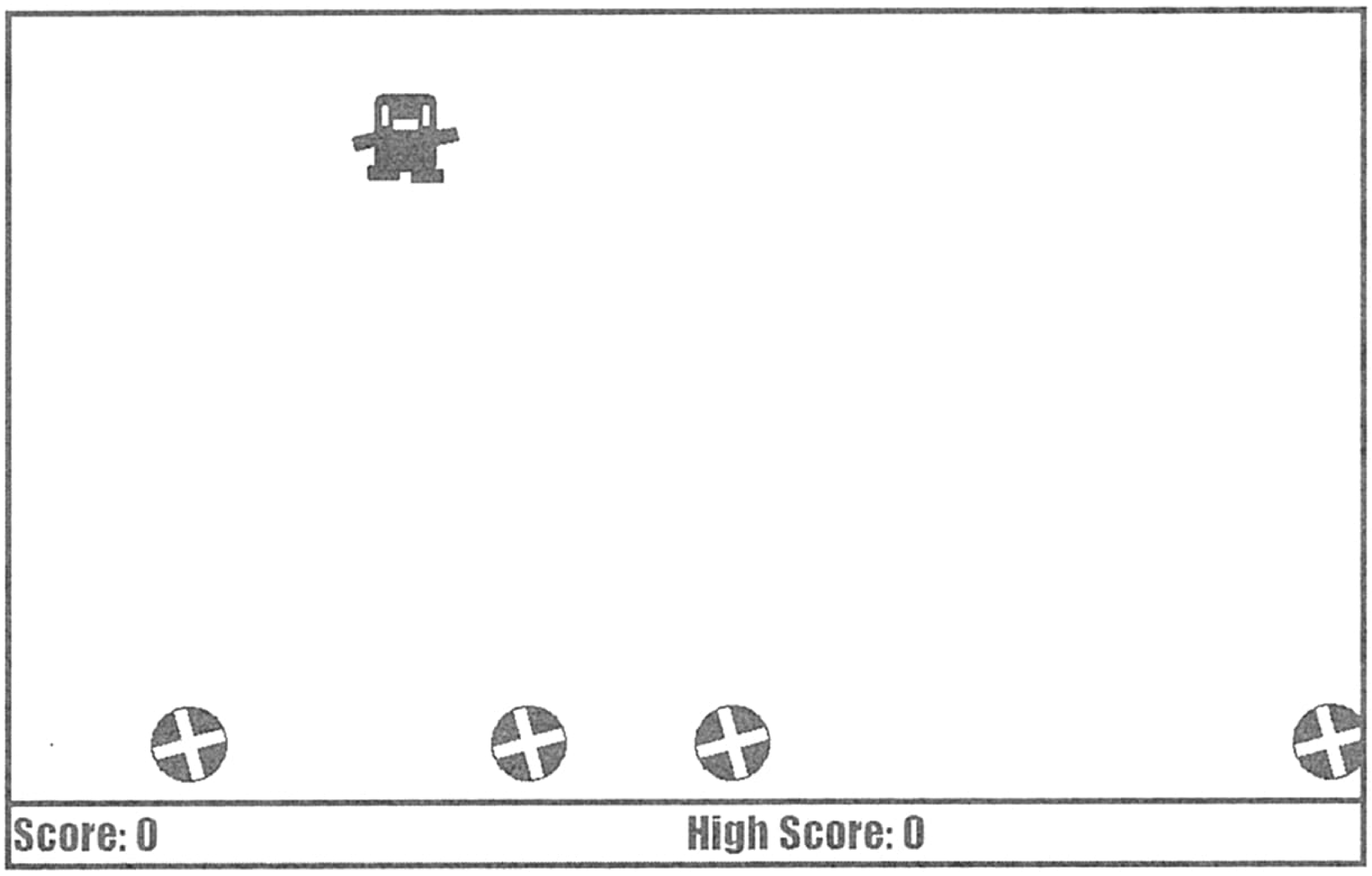
自機は左端に出現します(Fig.1)。カーソルキーの左右で、自機を左右に動かすことができます。画面の右側からは、左に向かって障害物(タル)が転がってきます。

▼ **Fig.1**: 障害物を越えるゲーム



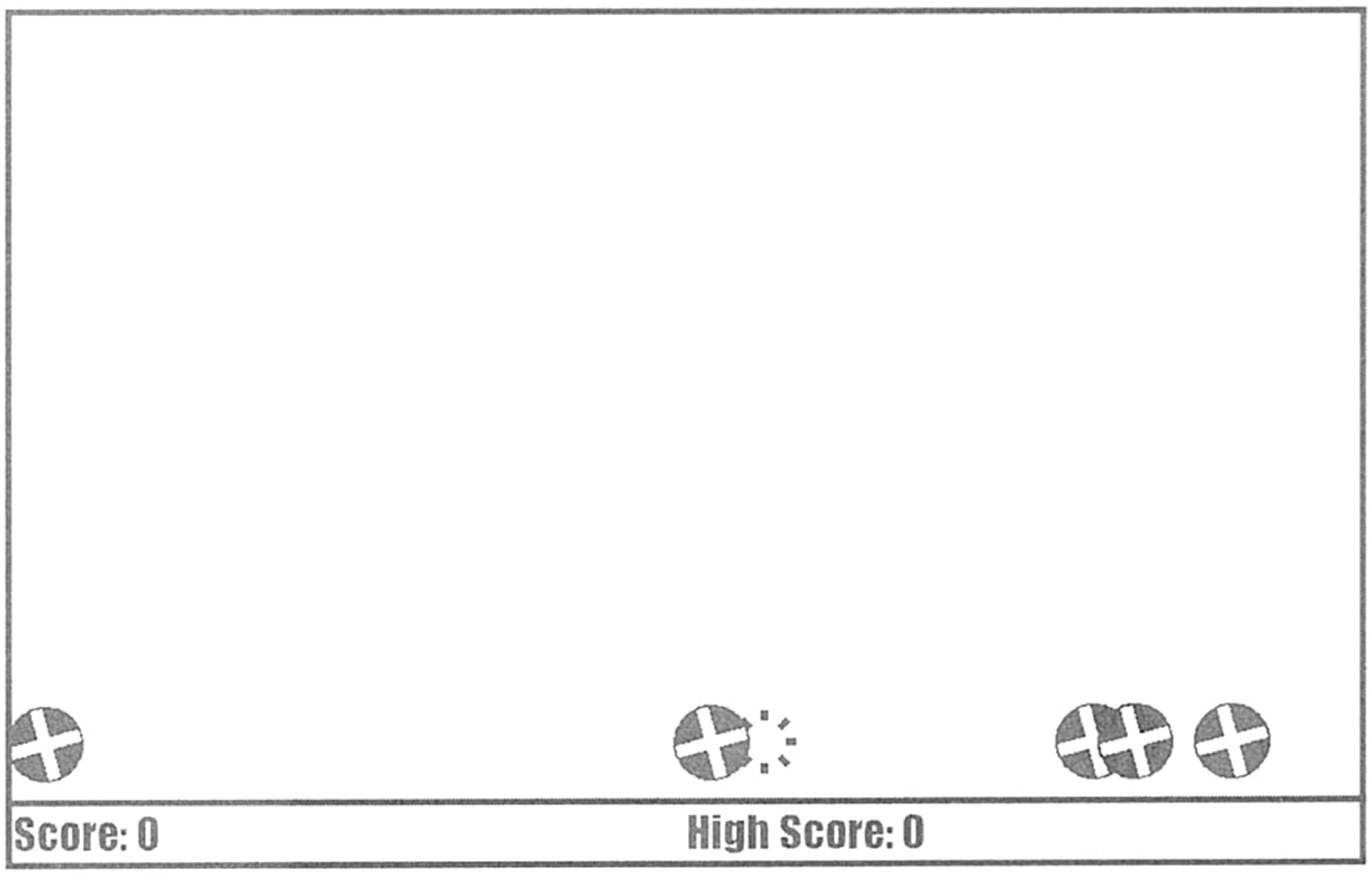
自機が地上にいるときにスペースキーを押すと、自機がジャンプします(Fig.2)。ジャンプ中でも、カーソルキーの左右を使って自機を左右に動かすことができます。自機は重力に引かれて落下しますが、上手に落下地点を選んで着地し、障害物をかわしてください。

▼ **Fig.2**: 自機をジャンプさせる



障害物に当たると、自機は消滅します(Fig.3)。消滅のアニメーションが終わると、自機は再び画面の左端に出現します。

▼ Fig.3: 障害物に当たると消滅する



本章の後半では、同じ内容のゲームをマウスで操作する例を紹介します(p.113)。また第4章では、同じゲームにスコア機能を付けて、より楽しめるゲームにします(p.126)。

キーボードの入力

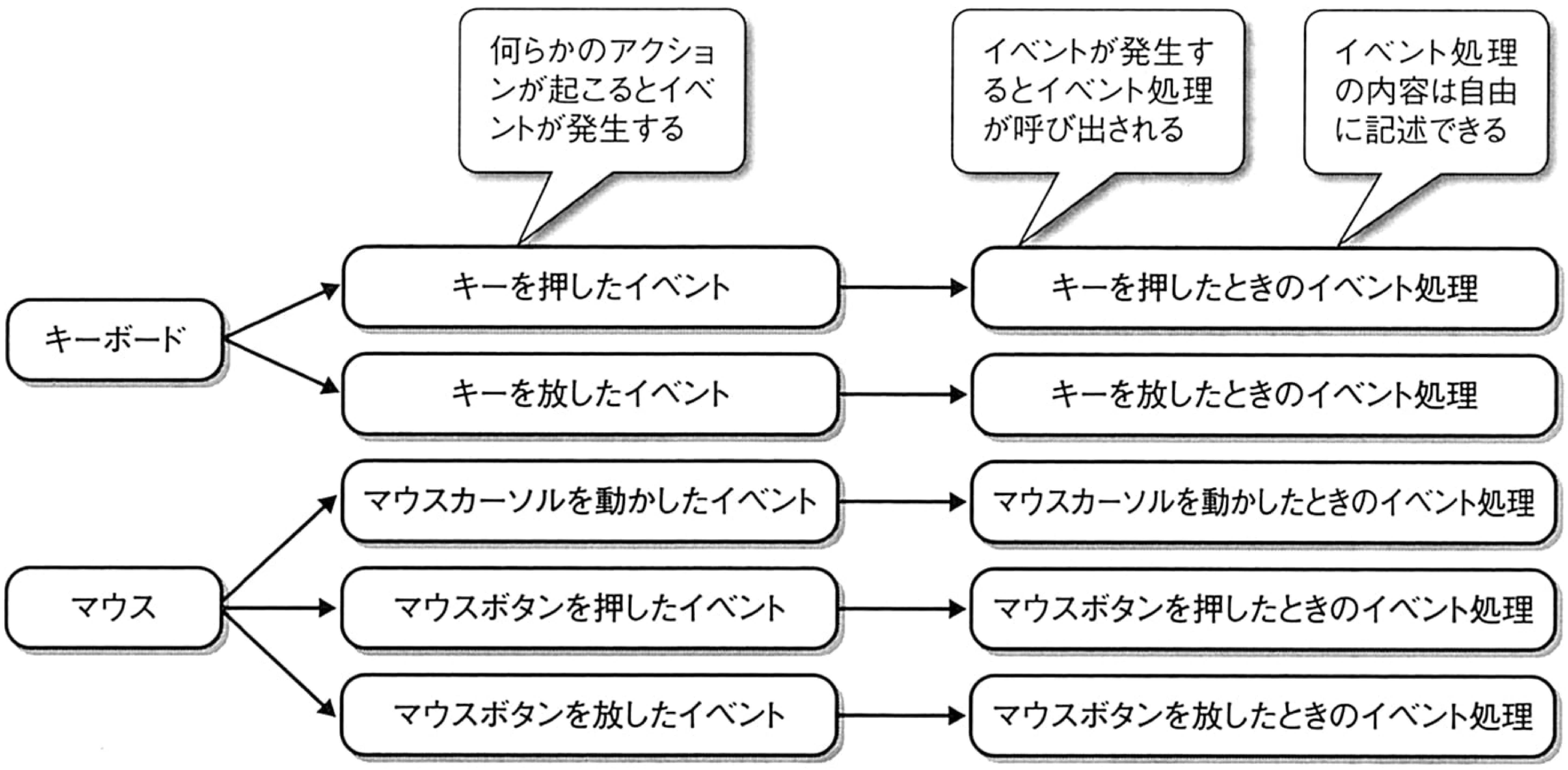
自機をキーボードで操作するためには、キーボードの入力を読み取る必要があります。入力を読み取る方法は開発環境によって異なりますが、JavaScriptなどの場合には、

- キーを押したとき
- キーを放したとき

に、イベントが発生します。イベントとは、何らかのアクションが起こったときに、プログラムに対して通知される信号です。イベントが発生するのは、キーボードやマウスから入力があったときや、ユーザーインターフェースのボタンやメニューが操作されたときなどです。

プログラム側では、発生したイベントに対して行う処理を適切に記述しておく必要があります。この処理をイベント処理と呼びます。イベントの種類に応じて、異なるイベント処理を記述することができます。イベントが発生すると、自動的にイベント処理が呼び出されます(Fig.4)。

▼ Fig.4: イベントとイベント処理



キーボードの入力に関して、JavaScriptには次のようなイベントがあります。

- **keydown**
  キーを押したときに発生
- **keypress**
  キーを押したときに発生(押し続けたときにも発生)
- **keyup**
  キーを放したときに発生

keydownとkeypressは似ていますが、キーを押し続けたときの挙動が違います。keydownは最初にキーを押したときに1回だけ発生しますが、keypressは最初の1回に加えて、キーを押し続けている間も一定時間ごとに発生します。

本書ではkeydownとkeyupを使います。これらのイベントは、押したり放したりしたキーのコード(番号)を通知します。例えばカーソルキーの左は37で、スペースキーは32です。

ゲームプログラムでは、指定したキーが押されているかどうかを、いつでも調べられると便利です。そこで、キーの状態を配列に保存することにしました。

keydownイベントが発生したら、押したキーのコードに対応する要素にtrueを書き込みます(Fig.5)。keyupイベントが発生したら、放したキーのコードに対応する要素にfalseを書き込みます(Fig.6)。

Fig.5: キーを押したとき

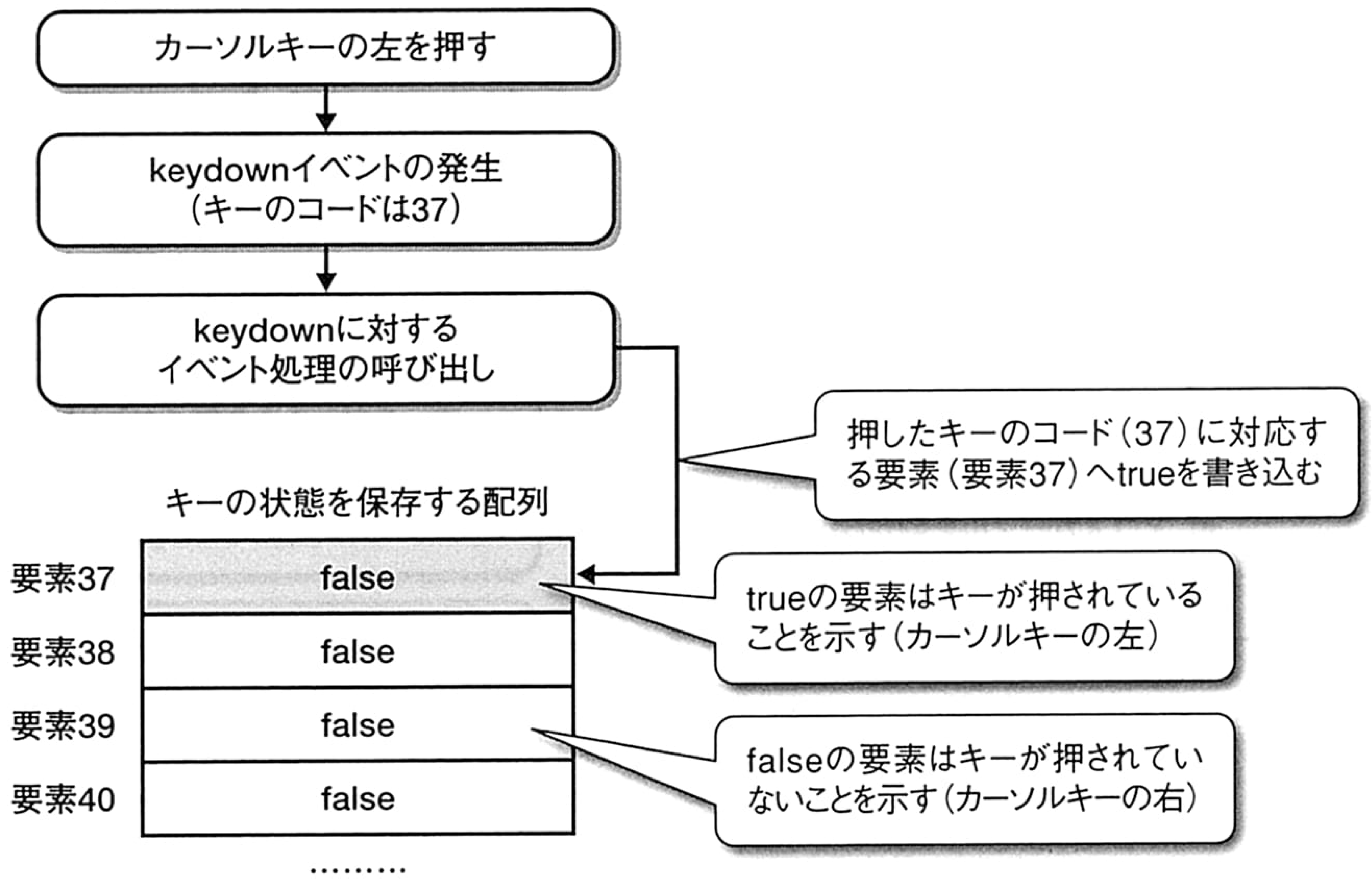
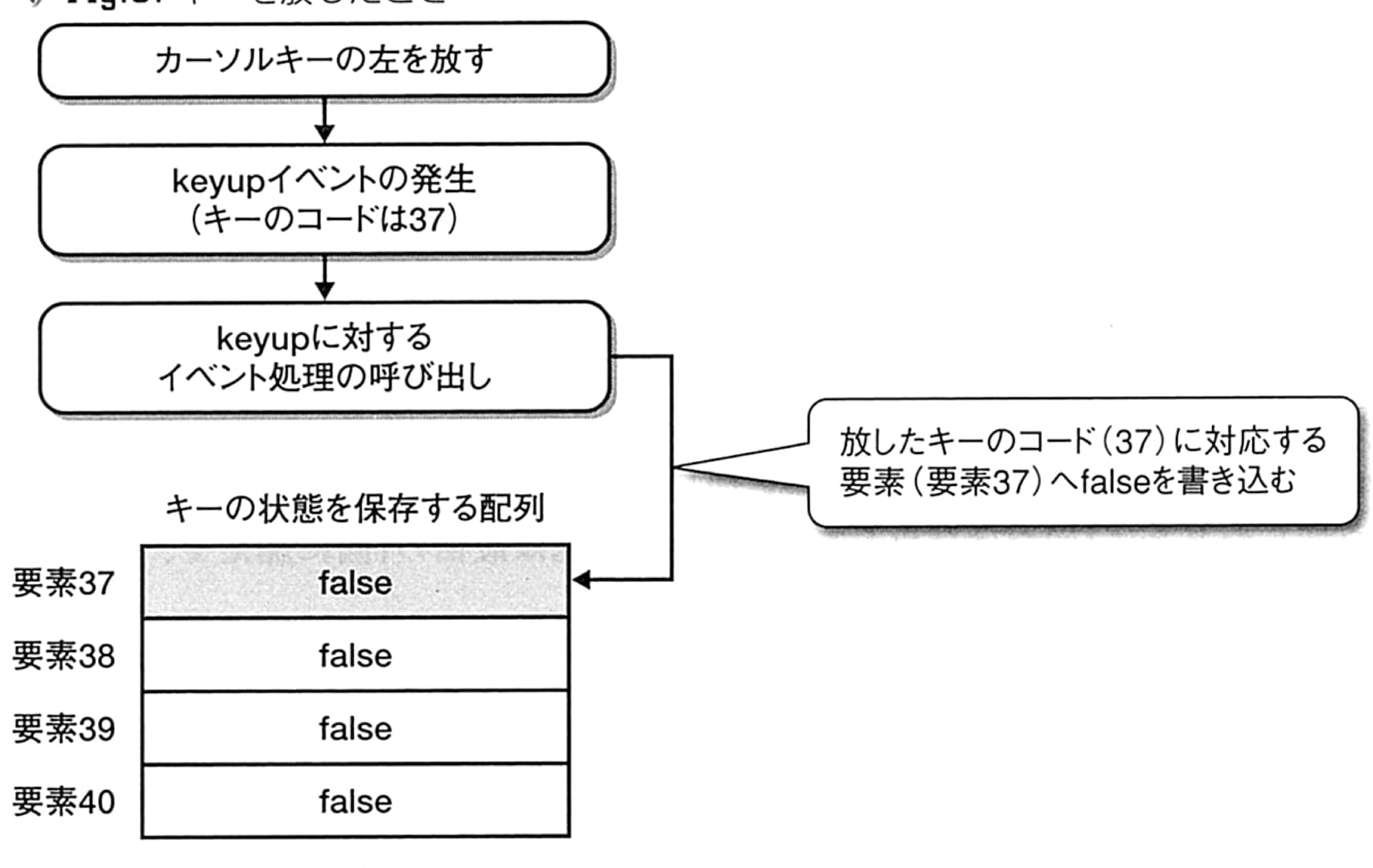


Fig.6: キーを放したとき



キーの状態が知りたいときには、キーのコードに対応する要素を調べます。trueならばキーが押されており、falseならば放されている（押されていない）ということです。

キーボードの入力を読み取るプログラム

キーボードの入力を読み取る処理は、色々なゲームから使うので、ライブラリ(common.js)に含めるとよいでしょう(List1)。Gameオブジェクトに対して、キーボードに関する処理を追加します。

List1では、

```
obj.key=new Array();
```

のように、キーの状態を表す配列を初期化します。最初は、

```
for (i=0; i<KEY_COUNT; i++) {
    obj.key[i]=false;
}
```

として、どのキーも押されていない状態にします。

keydownやkeyupに対するイベント処理を記述するには、documentオブジェクトのonkeydown関数やonkeyup関数を定義します。例えばonkeydown関数については、

```
document.onkeydown=function(e) {
```

のように書きます。引数のeはイベントオブジェクトと呼ばれ、キーのコードなど、イベントに関する情報を保持しています。

このイベントオブジェクトを使って、どのキーを押したのかを知るのですが、ここでちょっとした問題があります。実はブラウザによっては、引数のイベントオブジェクトが利用できないのです。そこで、

```
if (e) {
    obj.key[e.which]=true;
} else {
    obj.key[event.keyCode]=true;
}
```

のように、if文を使って場合分けを行います。イベントオブジェクトのeが有効な値の場合には、

```
obj.key[e.which]=true;
```

を実行します。e.whichはキーのコードです。eが無効な値の場合には、

```
obj.key[event.keyCode]=true;
```

を実行します。eventはイベントオブジェクトに相当するオブジェクトで、event.keyCodeはキーのコードです。

Firefoxなどでは、引数のeは有効な値ですが、eventオブジェクトが使えません。逆にInternet Explorerなどでは、eventオブジェクトは使えますが、引数のeは無効な値です。そのため、さまざまなブラウザに対応するためには、List1のような場合分けが必要になります。onkeyup関数についても同様です。

最後に、キーの状態を返すために、

```
obj.isKey=function(code) {
    return obj.key[code];
}
```

のようなisKey関数を定義します。コードで指定したキーが押されていればtrue、放されていればfalseを返します。

キーのコードもブラウザによって微妙に異なることがあるので、注意が必要です。List1では、ゲームで使うカーソルキーやスペースキーなどのコードを定数として用意しています。

▼ **List1**：キーボードの入力を読み取る(common.js)

```
// カーソルのキーコード
var KEY_LEFT=37;
var KEY_RIGHT=39;
var KEY_UP=38;
var KEY_DOWN=40;

// スペースキー、ENTERキー、ESCキーのキーコード
var KEY_SPACE=32;
var KEY_ENTER=13;
var KEY_ESC=27;

// キーの状態を表す配列のサイズ
var KEY_COUNT=256;

// ゲームオブジェクト
function Game() {
  // ...(中略)...
```

```
  // キーの状態を表す配列を初期化する
  obj.key=new Array();
  for (i=0; i<KEY_COUNT; i++) {

    // 最初はどのキーも押されていない状態にする
    obj.key[i]=false;
  }

  // キーを押したときのイベント処理
  document.onkeydown=function(e) {

    // 引数がある場合の処理(Firefoxなど)
    if (e) {
      obj.key[e.which]=true;
    } else

    // 引数がない場合の処理(Internet Explorerなど)
    {
      obj.key[event.keyCode]=true;
    }
  }

  // キーを放したときのイベント処理
  document.onkeyup=function(e) {

    // 引数がある場合の処理(Firefoxなど)
    if (e) {
      obj.key[e.which]=false;
    } else

    // 引数がない場合の処理(Internet Explorerなど)
    {
      obj.key[event.keyCode]=false;
    }
  }

  // キーの状態を取得する関数
  // 指定したキーが押されているときにtrueを返す
  obj.isKey=function(code) {
    return obj.key[code];
  }
}
```

当たり判定処理

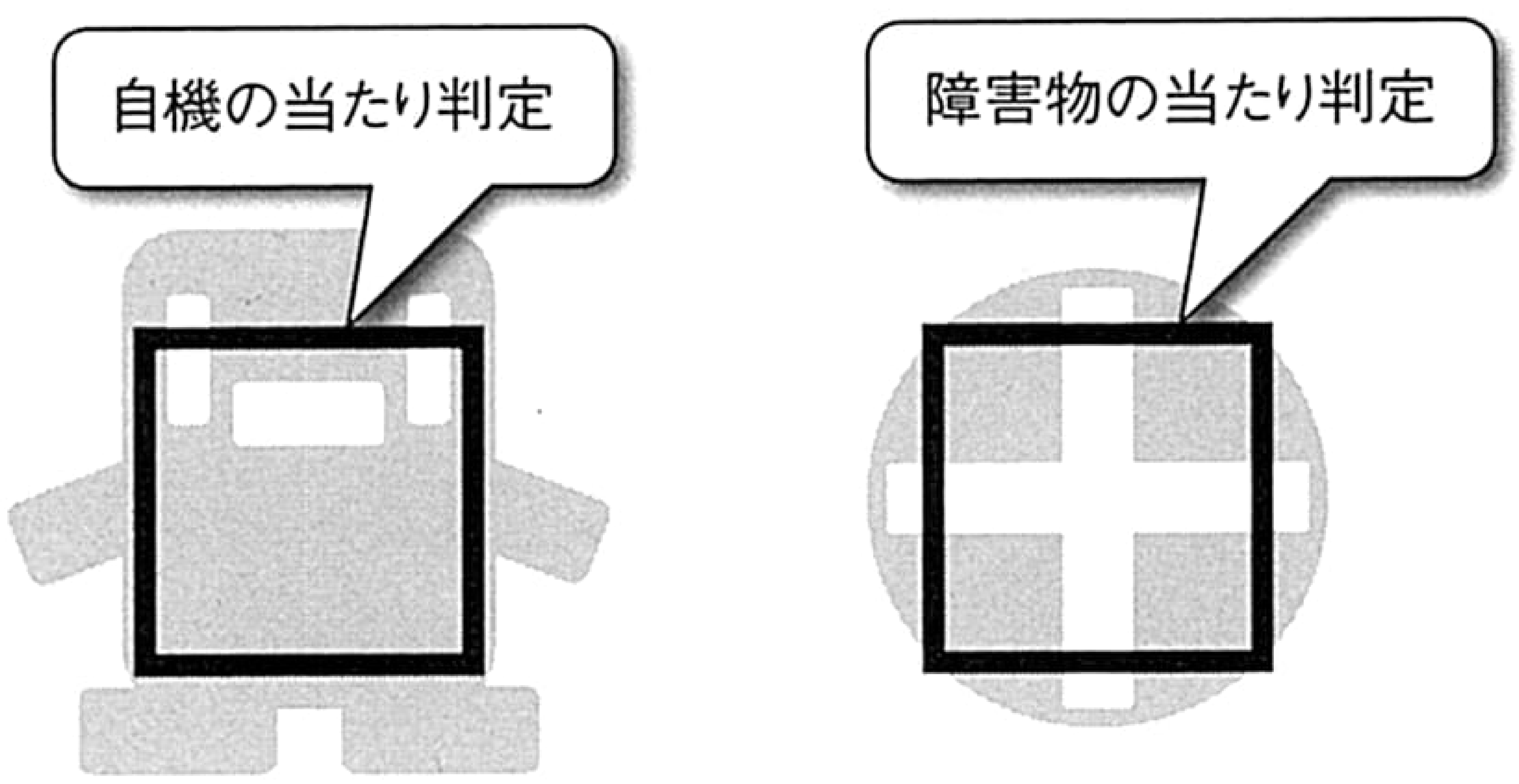
サンプルにおいて、障害物に接触した自機を消滅させるためには、自機が障害物に接触したかどうかを調べる必要があります。それには、当たり判定処理(あたりはんていしょり)と呼ばれる処理を行います。

当たり判定処理とは、主にゲームプログラムにおいて、物体同士が接触したかどうかを判断する処理のことです。例えば自機と敵などについて当たり判定処理を適用した結果、接触したことが分かったら、自機を爆発させる処理などを行います。

当たり判定処理には色々な方法がありますが、ここでは矩形を使った方法を紹介します。2次元グラフィックスのゲームでは、矩形や円を使って当たり判定処理を行うのが一般的です。

2つの物体について、当たり判定処理を行うことを考えましょう。まず各々の物体について、矩形の当たり判定を定義します(Fig.7)。当たり判定とは、物体が他の物体と接触する可能性のある部分のことです。

Fig.7: 当たり判定



現実の物体では、当たり判定は物体の見た目通りですが、ゲームでは見た目と当たり判定が異なることがよくあります。その理由は次の通りです。

- **当たり判定処理を高速にする**
  複雑な形をした物体について、見た目通りの複雑な当たり判定を定義すると、当たり判定処理に時間がかかります。多くの場合には、精密に物体の形に合わせた当たり判定は必要なく、もっと単純な形の当たり判定で十分です。
- **ゲームを遊びやすくする**
  当たり判定の大小を調整することによって、ストレスなく遊べるゲームになります。例えば自機の当たり判定を小さめに設定すると、敵や障害物に当たりにくくなり、プレイヤーが「避けたはずなのに当たった」と思うことが少なくなります。また逆に、武器の当たり判定を大きめに設定すると、敵に武器が当たりやすくなり、「当てたはずなのに外れた」と感じることが減ります。

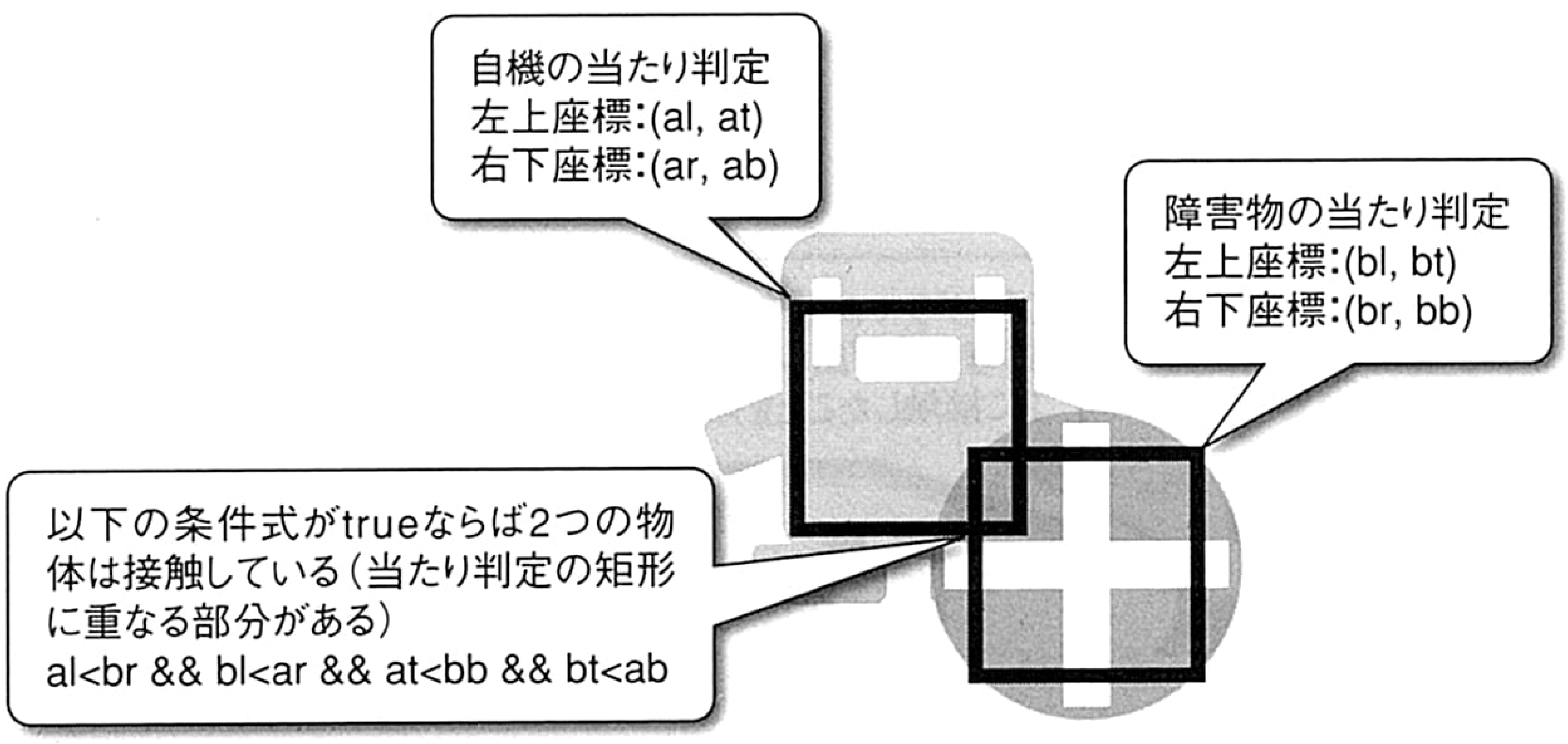
次に、定義した当たり判定を使って、当たり判定処理を行います(Fig.8)。矩形の当たり判定を使う場合には、2つの矩形が重なったら、物体同士が接触したと判定します。

2つの矩形の左上の座標を(al, at)および(bl, bt)、右下の座標を(ar, ab)および(br, bb)とすると、矩形同士が重なる条件式は、

```
al<br && bl<ar && at<bb && bt<ab
```

となります。条件式が成立すれば(trueになれば)、2つの矩形に重なる部分があるということです。

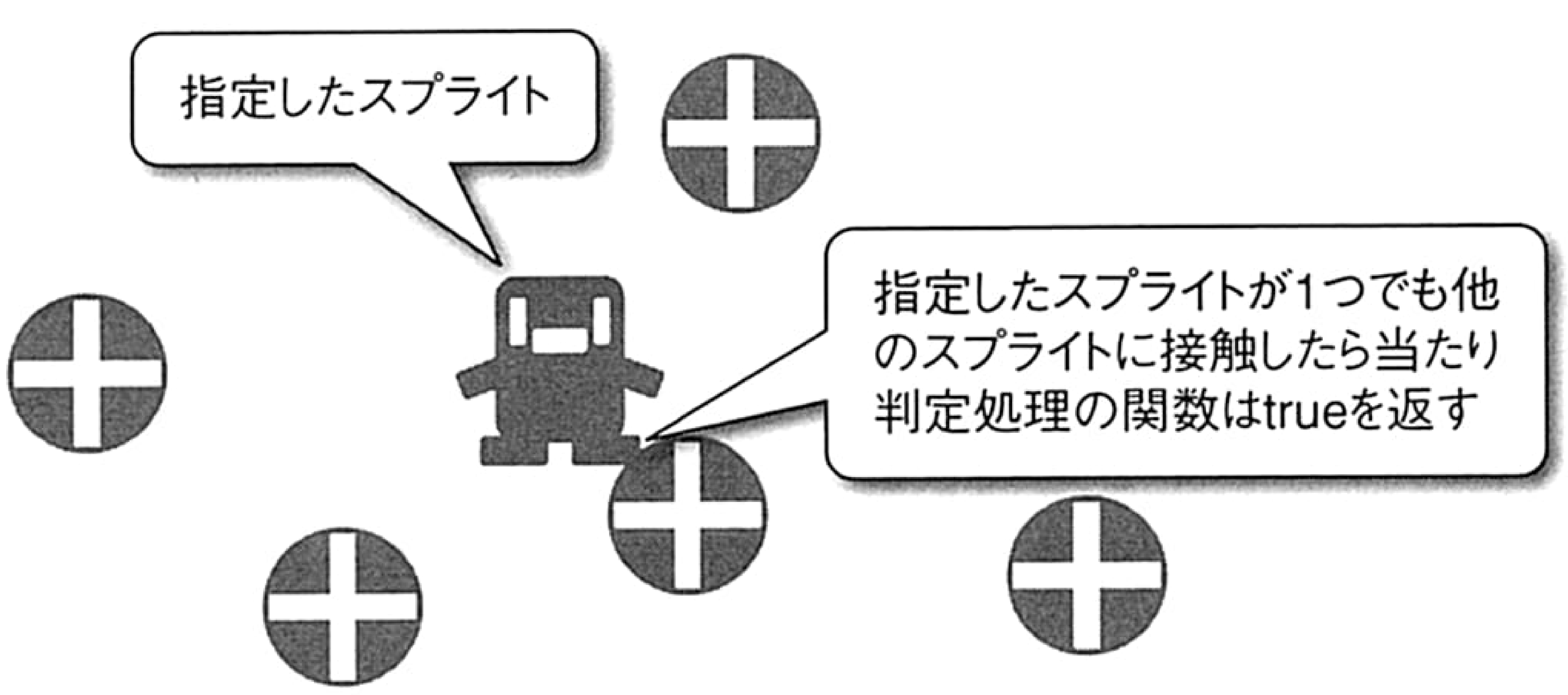
▼ Fig.8: 当たり判定処理



スプライトの当たり判定処理

本書のサンプルではスプライトを使うので、スプライト同士の当たり判定処理を行うことにしました。指定したスプライトが、他のスプライトのいずれかに接触したかどうかを調べる関数を用意します(Fig.9)。

▼ Fig.9: スプライトの当たり判定処理

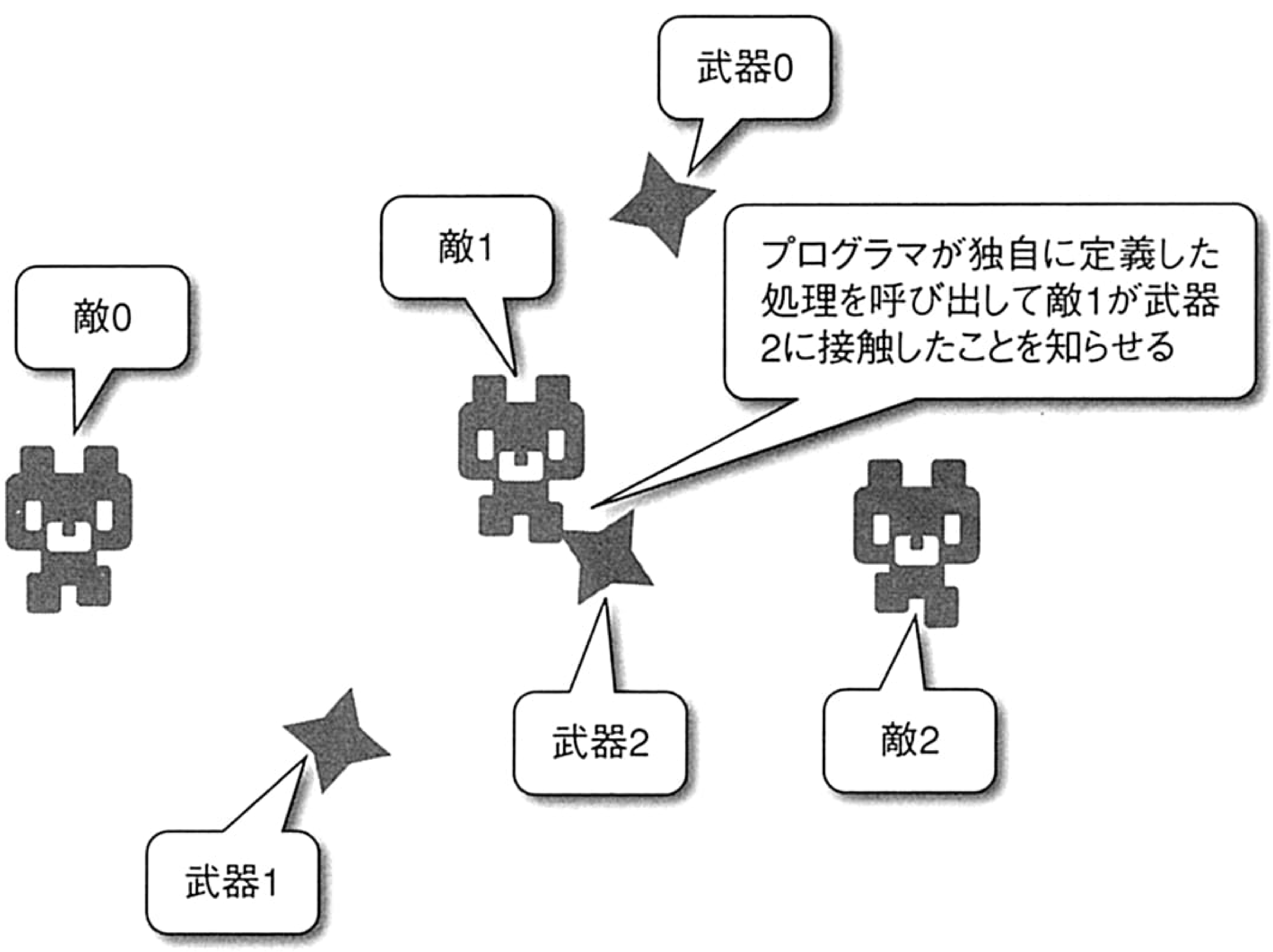


関数は1つでも他のスプライトに接触したらtrueを返し、1つも接触しなかったらfalseを返します。障害物に接触したら自機を消滅させる、といった処理には、この関数だけで十分です。

一方、武器が敵に接触したら武器と敵の両方を消滅させる、といった処理には、この関数だけでは不十分です。いずれかのスプライトに接触したかどうかだけではなくて、どのスプライトに接触したのかを知る必要があります。

そこで、他のスプライトに接触するたびに、プログラマが独自の処理を呼び出せる仕組みを用意します(Fig.10)。この仕組みを利用すると、例えば武器がどの敵に接触したのかが分かるので、武器と敵の両方を消滅させたり、敵の種類に応じてスコアを加算したりといったことができます。

Fig.10: 接触時に独自の処理を呼び出す



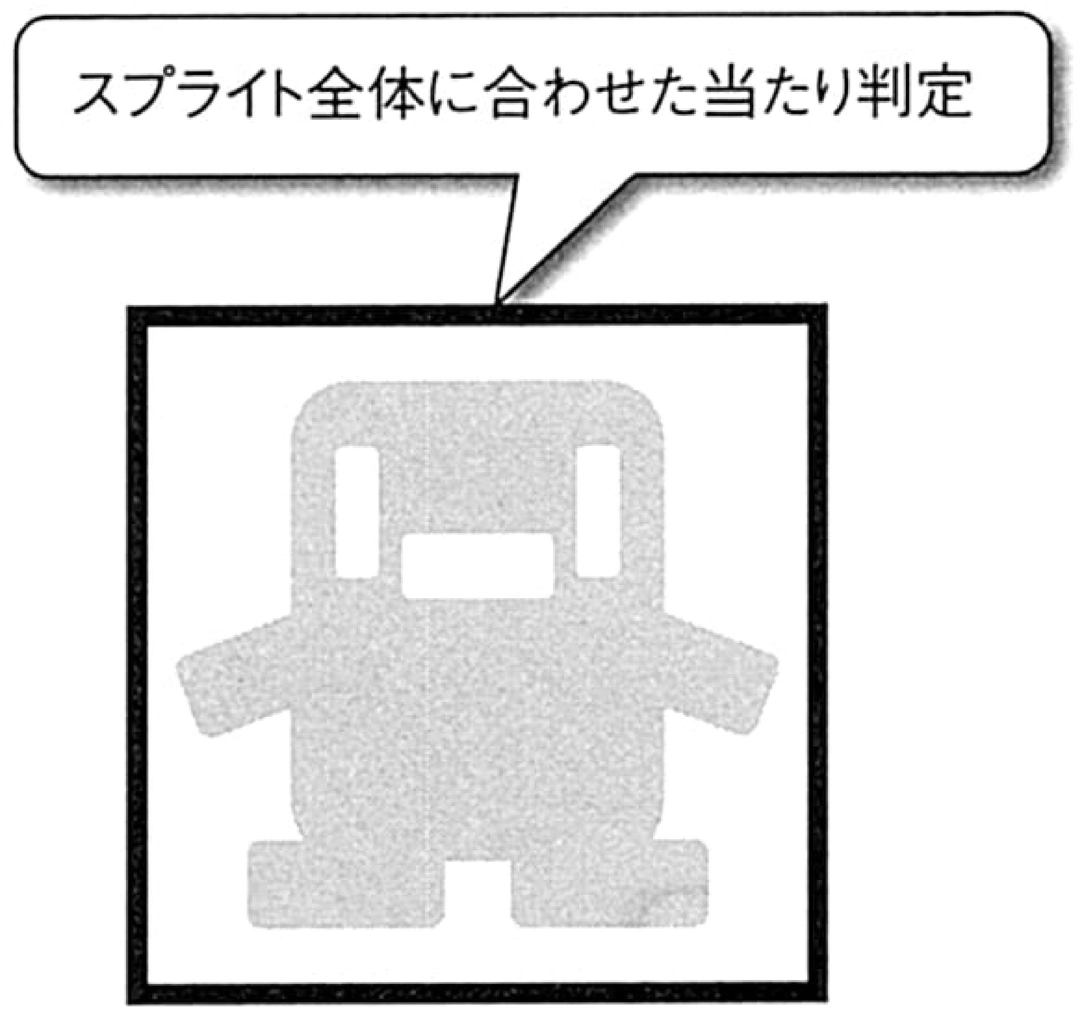
当たり判定処理のプログラム

当たり判定処理のプログラムは、SpriteオブジェクトとGameオブジェクトに追加します(List2)。まずSpriteオブジェクトでは、

```
obj.hit=new Array(0, 0, 1, 1);
```

のように、当たり判定を表すhitプロパティを定義します。hitプロパティには、当たり判定の左･上･右･下の座標を配列で格納することにしました。初期状態では、当たり判定をスプライトの全体に合わせています(Fig.11)。

▼ **Fig.11**：初期状態の当たり判定



当たり判定は、自機や敵といったスプライトの種類によって違います。そこで、

```
obj.initHit=function(array) {
    obj.hit=array;
}
```

のように、当たり判定を自由に設定するためのinitHit関数を用意しました。
次の、

```
obj.onHitSprite=function(spr) {};
```

は、他のスプライトに接触したときに呼び出すonHitSprite関数です。この関数の内容を定義することにより、ゲームごとに独自の処理を行うことができます。
実際に当たり判定処理を行うのは、

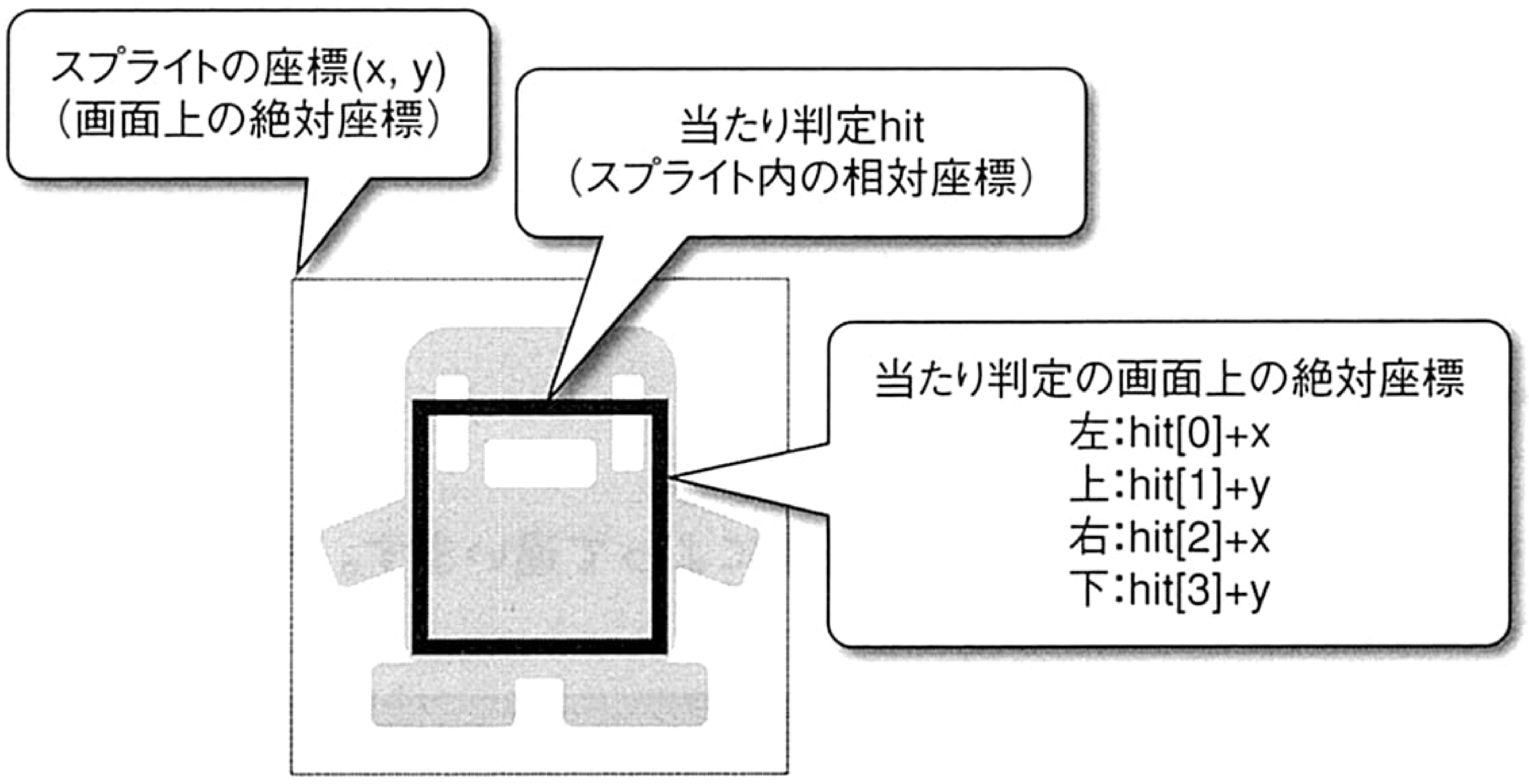
```
obj.testHitSprite=function(spr) {
```

のようなtestHitSprite関数です。この関数は、変数objが表すスプライトと、引数sprが表すスプライトの間で、当たり判定処理を行います。まず、

```
var a=obj.hit, b=spr.hit;
var al=a[0]+obj.x, ar=a[2]+obj.x;
var at=a[1]+obj.y, ab=a[3]+obj.y;
var bl=b[0]+spr.x, br=b[2]+spr.x;
var bt=b[1]+spr.y, bb=b[3]+spr.y;
```

のように、スプライトの座標と当たり判定の座標を使って、画面上の当たり判定の座標を求めます(Fig.12)。hitプロパティに格納した当たり判定の座標は、スプライト内の相対座標なので、画面上の絶対座標にする必要があるためです。

▼ **Fig.12**: 当たり判定の座標を求める



座標を求めたら、

```
if (al<br && bl<ar && at<bb && bt<ab) {
```

のように当たり判定処理を行います。そして、スプライト同士が接触したら、

```
obj.onHitSprite(spr);
```

のように前述のonHitSprite関数を呼び出し、trueを返します。接触しなかったら、falseを返します。
Gameオブジェクトでは、

```
obj.testHitSprite=function(spr) {
```

のようなtestHitSprite関数を定義します。この関数は、引数sprで指定されたスプライトと、他の全てのスプライトの間で、当たり判定処理を行います。1つでも他のスプライトに接触していたらtrueを返し、1つも接触していなかったらfalseを返します。
当たり判定処理は、

```
for (i=0; i<obj.sprite.length; i++) {
    if (spr!=obj.sprite[i]) {
        if (spr.testHitSprite(obj.sprite[i])) {
```

```
            hit=true;
        }
    }
}
```

のように行います。spriteプロパティは全スプライトを保持する配列です(p.64)。この配列を使って、指定されたスプライトと、他の全てのスプライトの組み合わせについて、先に述べたSpriteオブジェクトのtestHitSprite関数を呼び出します。

List2: 当たり判定処理のプログラム(common.js)

```
// スプライト
function Sprite(div, zIndex) {

  // 当たり判定
  obj.hit=new Array(0, 0, 1, 1);

  // 当たり判定の設定
  obj.initHit=function(array) {
    obj.hit=array;
  }

  // 他のスプライトに接触したときに呼び出す処理
  obj.onHitSprite=function(spr) {};

  // 他のスプライトとの当たり判定処理
  obj.testHitSprite=function(spr) {

    // 2つのスプライトについて当たり判定の上下左右の座標を求める
    var a=obj.hit, b=spr.hit;
    var al=a[0]+obj.x, ar=a[2]+obj.x;
    var at=a[1]+obj.y, ab=a[3]+obj.y;
    var bl=b[0]+spr.x, br=b[2]+spr.x;
    var bt=b[1]+spr.y, bb=b[3]+spr.y;

    // 2つのスプライトの座標を調べて、
    // 当たり判定が接触しているかどうかを判断する
    if (al<br && bl<ar && at<bb && bt<ab) {

      // 接触したら、接触したときに行う処理を呼び出す
      obj.onHitSprite(spr);

      // 接触したらtrueを返す
      return true;
    }

    // 接触しなかったらfalseを返す
```

```
    return false;
  }
}

// ゲーム本体
function Game() {

  // スプライト同士の当たり判定処理
  obj.testHitSprite=function(spr) {
    var hit=false;
    var i;

    // 引数で指定したスプライト(spr)と、
    // 他の全てのスプライトとの間で、
    // 当たり判定処理を行う
    for (i=0; i<obj.sprite.length; i++) {
      if (spr!=obj.sprite[i]) {

        // Spriteオブジェクトの当たり判定処理を呼び出す
        if (spr.testHitSprite(obj.sprite[i])) {
          hit=true;
        }
      }
    }

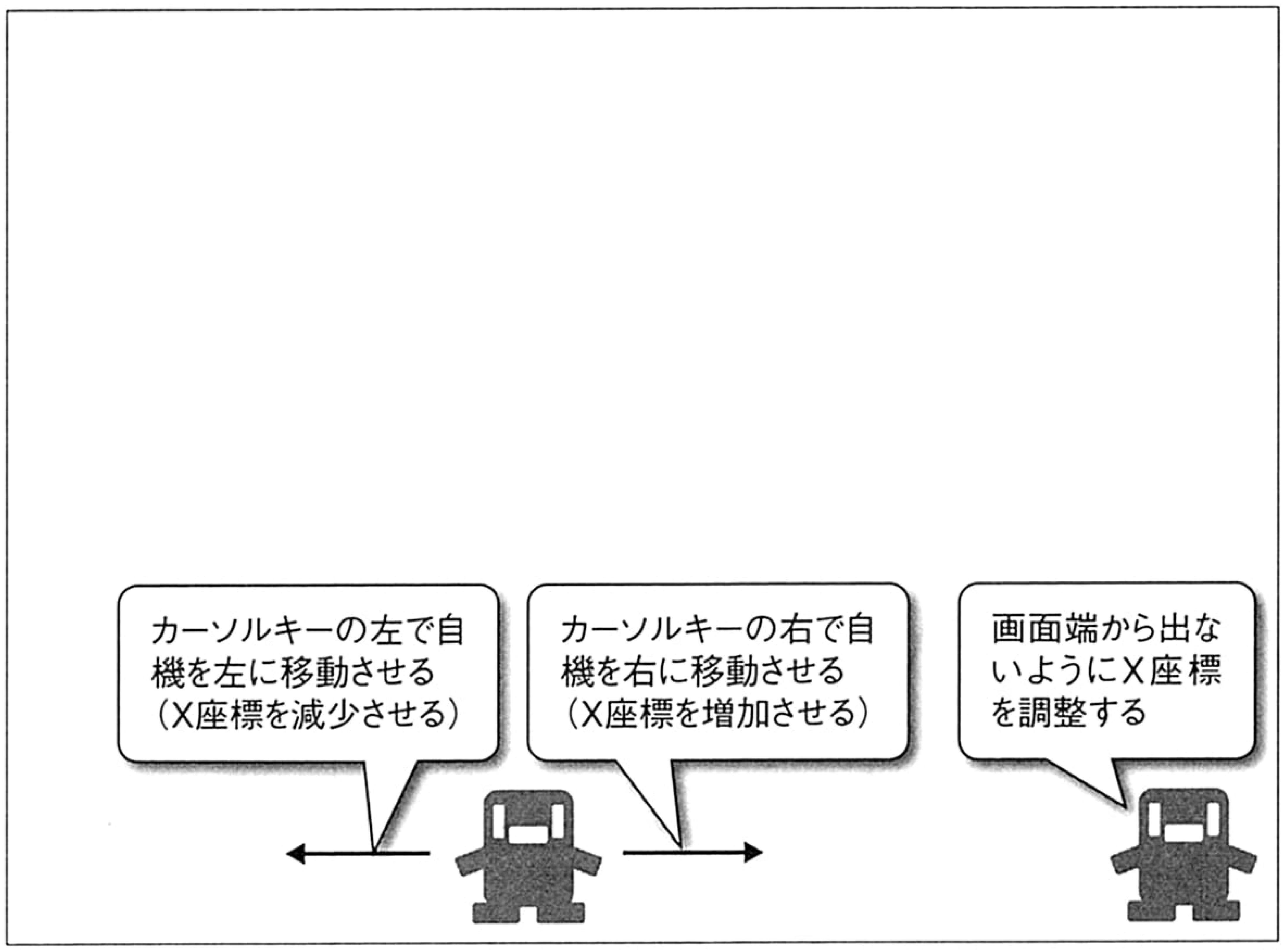
    // 1つでも他のスプライトに接触したらtrueを返す
    return hit;
  }
}
```

キーボードで自機を動かす

これまでに作成したキーボードの入力処理と当たり判定処理を使えば、ジャンプで障害物を避ける自機のプログラムを書くことができます。その前に、自機を動かす方法を詳しく検討してみましょう。

カーソルキーの左右を入力したら、自機を左右に移動させます(Fig.13)。これは自機のX座標を左右に変化させることで実現します。自機が画面端からはみ出しそうになったら、出ないように修正します。

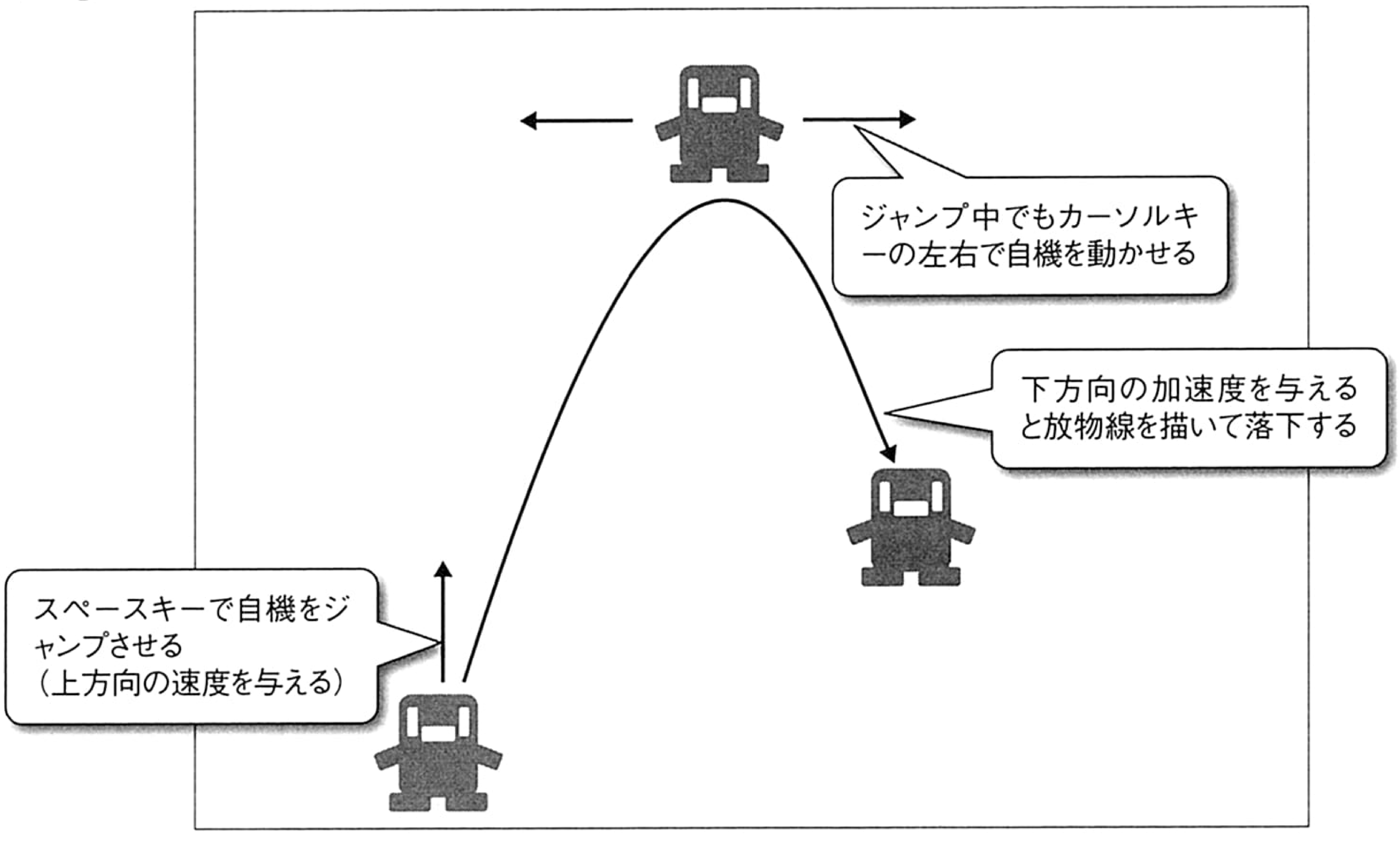
▼ **Fig.13**: 左右への移動



自機がジャンプしていないときに、スペースキーを入力したら、自機をジャンプさせます(Fig.14)。ジャンプさせるには、自機のY座標に上方向の速度を加算します。同時に、速度に下方向の加速度を加算すると、自機は放物線を描いて落下します。

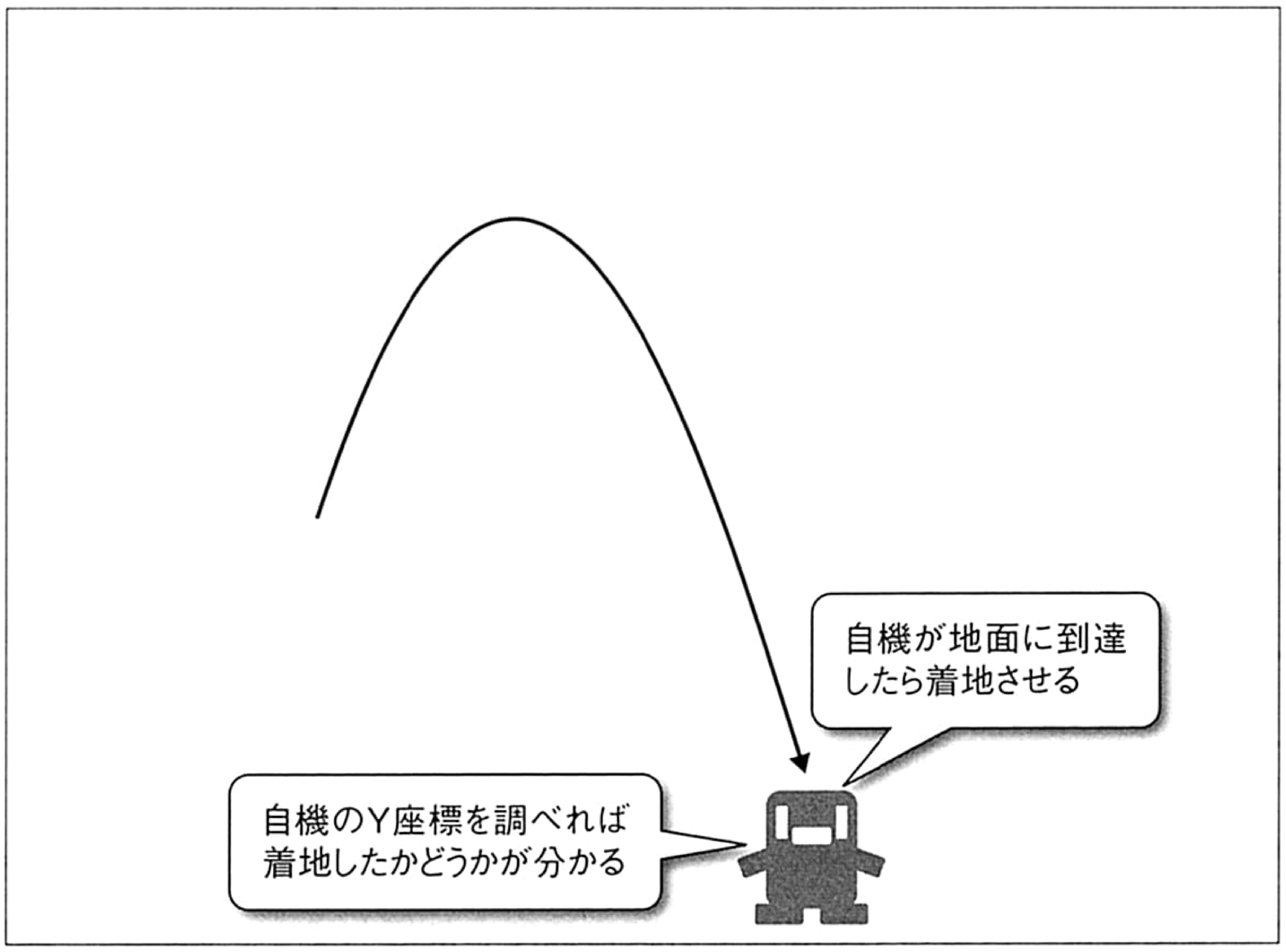
このサンプルでは、滞空中でも自機を左右に動かせるようにしました。これは現実のジャンプとは違いますが、障害物が避けやすいのでゲームが面白くなります。自機を左右に動かすには、地上にいるときと同じくX座標を左右に変化させます。

▼ **Fig.14**: ジャンプ



自機が落下して地面に到達したら、着地させます(Fig.15)。地面に到達したかどうかは、自機のY座標を調べれば分かります。着地した自機は、再びジャンプできるようになります。

▼ Fig.15: 着地

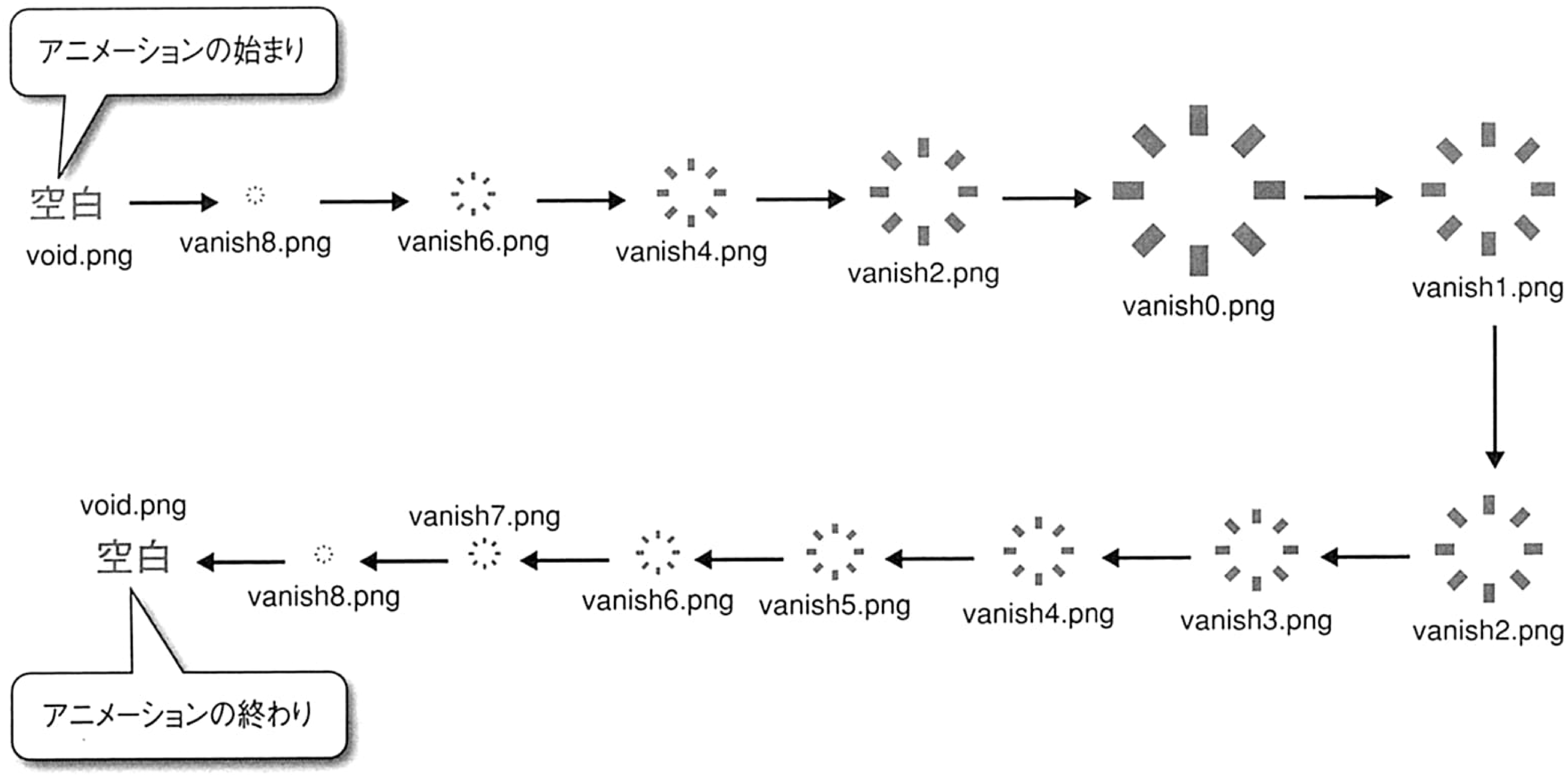


障害物に接触したかどうかを判断するためには、自機と全ての障害物の間で、当たり判定処理を行います。接触したら、自機のアニメーションを消滅のアニメーションに切り替えます(Fig.16)。消滅のアニメーションは15コマです(Fig.17)。

▼ Fig.16: 消滅



▼ **Fig.17**: 消滅のアニメーション



消滅のアニメーションが終わったら、ゲームを初期化します。自機を左端に戻し、障害物を再び出現させて、新たにゲームが始められるようにします。

キーボードで自機を動かすプログラム

検討した方法をもとに、自機を動かすプログラムを書きます(List3)。自機を表すPlayerオブジェクトを定義します。

まずスプライトを作成し、アニメーションを設定します。次に、

```
obj.sprite.initHit(playerHit);
```

として、当たり判定を設定します。当たり判定を表す配列playerHitは、

```
playerHit=new Array(0.25, 0.25, 0.75, 0.75);
```

のように定義しました(p.111のList5)。これは自機の見た目よりも少し小さめの当たり判定になっています(Fig.18)。

▼ **Fig.18**: 自機の当たり判定



自機には、ジャンプしている状態と、地上にいる状態があります。この2つの状態を区別するために、

```
obj.jump=false;
```

のようなjumpプロパティを用意します。trueのときはジャンプ中で、falseのときは地上にいます。

また、自機には通常の状態(キーボードで操作できる状態)と、消滅中の状態があります。これは、

```
obj.state=STATE_ALIVE;
```

のように、stateプロパティで管理することにしました。一定時間ごとに実行するonUpdate関数では、

```
obj.sprite.onUpdate=function() {
    switch (obj.state) {
        case STATE_ALIVE: obj.alive(); break;
        case STATE_DEAD: obj.dead(); break;
    }
}
```

のように、stateプロパティの値に応じて分岐します。switch文は値に応じて処理を分岐させる構文です。

自機が通常の状態のときには、alive関数を実行します。この関数では、

```
if (game.isKey(KEY_LEFT)) x-=PLAYER_VX;
if (game.isKey(KEY_RIGHT)) x+=PLAYER_VX;
```

のように、カーソルキーの入力に応じて自機を左右に移動させます。
　地上にいるときにスペースキーを入力したら、

```
if (game.isKey(KEY_SPACE)) {
    vy=PLAYER_VY;
    obj.jump=true;
}
```

のように、上方向の速度を設定し、ジャンプ状態にします。
　ジャンプ状態のときは、

```
y+=vy;
vy+=PLAYER_AY;
```

のように、座標に速度を加算し、速度に加速度を加算します。そして、

```
if (y>=YMAX) {
    y=YMAX;
    obj.jump=false;
}
```

のようにY座標が一定値以上になったら、着地したと判定し、ジャンプしていない状態に戻ります。
　障害物との当たり判定処理は、

```
if (game.testHitSprite(obj.sprite)) {
    obj.sprite.initAnim(vanishAnim);
    obj.state=STATE_DEAD;
}
```

のように行います。1つでも障害物に接触したら、消滅のアニメーション(vanishAnim)に切り替えるとともに、stateプロパティを消滅状態にします。
　消滅状態の処理は、

```
obj.dead=function() {
    obj.sprite.stepAnim();
    if (obj.sprite.isAnimOver()) {
        init();
    }
}
```

のようなdead関数で行います。消滅のアニメーションを再生し、アニメーションが終わったら、init関数(p.111のList5)を呼び出してゲームを初期化します。

List3: 自機のオブジェクト(jumpKey.js)

```
// 自機の速度、加速度
var PLAYER_VX=0.2;
var PLAYER_VY=-1.0;
var PLAYER_AY=0.1;

// 自機の状態
var STATE_ALIVE=0;
var STATE_DEAD=1;

// 自機のオブジェクト
function Player() {
  var obj=this;

  // スプライトの作成
  obj.sprite=game.createSprite();

  // アニメーションの設定
  obj.sprite.initAnim(playerWalkAnim);

  // 当たり判定の設定
  obj.sprite.initHit(playerHit);

  // 座標と速度の設定
  obj.sprite.setXY(0, YMAX);
  obj.vy=0;

  // ジャンプしているかどうか
  obj.jump=false;

  // 自機の状態
  obj.state=STATE_ALIVE;

  // ゲーム独自の更新処理
  obj.sprite.onUpdate=function() {

    // 状態に応じて分岐する
    switch (obj.state) {
      case STATE_ALIVE: obj.alive(); break;
      case STATE_DEAD: obj.dead(); break;
    }
  }

  // 通常状態の処理
```

```
obj.alive=function() {

  // 座標と速度を取得する
  var x=obj.sprite.getX(), y=obj.sprite.getY();
  var vy=obj.vy;

  // カーソルキーで左右に移動する
  if (game.isKey(KEY_LEFT)) x-=PLAYER_VX;
  if (game.isKey(KEY_RIGHT)) x+=PLAYER_VX;

  // 画面からはみ出したら座標を修正する
  if (x<0) x=0;
  if (x>XMAX) x=XMAX;

  // ジャンプしていないときの処理
  if (!obj.jump) {

    // スペースキーでジャンプする
    if (game.isKey(KEY_SPACE)) {
      vy=PLAYER_VY;
      obj.jump=true;
    }

    // 左右に移動したら、
    // 歩くアニメーションを再生する
    if (x!=obj.sprite.getX()) {
      obj.sprite.stepAnim();
    }
  } else

  // ジャンプしているときの処理
  {
    // 座標と速度を更新する
    y+=vy;
    vy+=PLAYER_AY;

    // 画面の下端に達したら着地する
    if (y>=YMAX) {
      y=YMAX;
      obj.jump=false;
    }
  }

  // スプライトの座標を更新する
  obj.sprite.setXY(x, y);

  // 速度のプロパティを更新する
  obj.vy=vy;
```

```
    // 障害物に接触したときの処理
    if (game.testHitSprite(obj.sprite)) {

      // 消滅のアニメーションに切り替える
      obj.sprite.initAnim(vanishAnim);

      // 消滅状態にする
      obj.state=STATE_DEAD;
    }
  }

  // 消滅状態の処理
  obj.dead=function() {

    // アニメーションを再生する
    obj.sprite.stepAnim();

    // アニメーションが終了したら、
    // ゲームを最初から始める
    if (obj.sprite.isAnimOver()) {
      init();
    }
  }
}
```

switch文

switch文は値に応じて処理を振り分けるための構文です。記法は、

```
switch (式) {
    case 値A:
        処理A;
        break;
    case 値B:
        処理B;
        break;
    default:
        処理X;
        break;
}
```

となっています。式が値Aのときには処理Aを、値Bのときには処理Bを実行します。式がどの値とも一致しないときには、default以下の処理Xを実行します。caseは必要に応じて、いくつでも並べることができます。上記のswitch文と同じ内容を、

```
if (値A) {
    処理A;
} else
if (値B) {
    処理B;
} else {
    処理X;
}
```

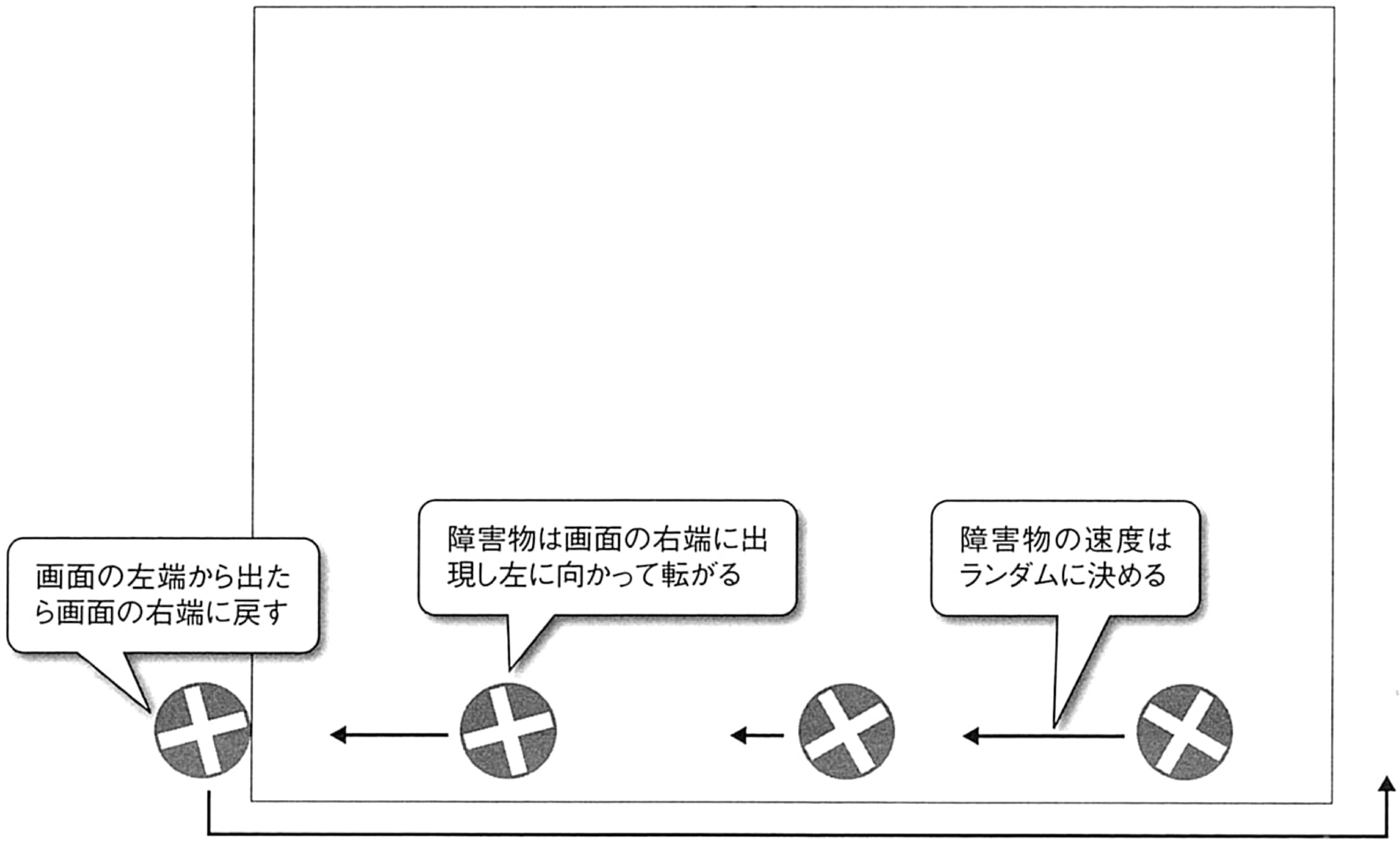
のように、if文を使って書くこともできます。どちらを使っても構いませんが、プログラムが簡潔に書ける方を選ぶとよいでしょう。

障害物を動かす

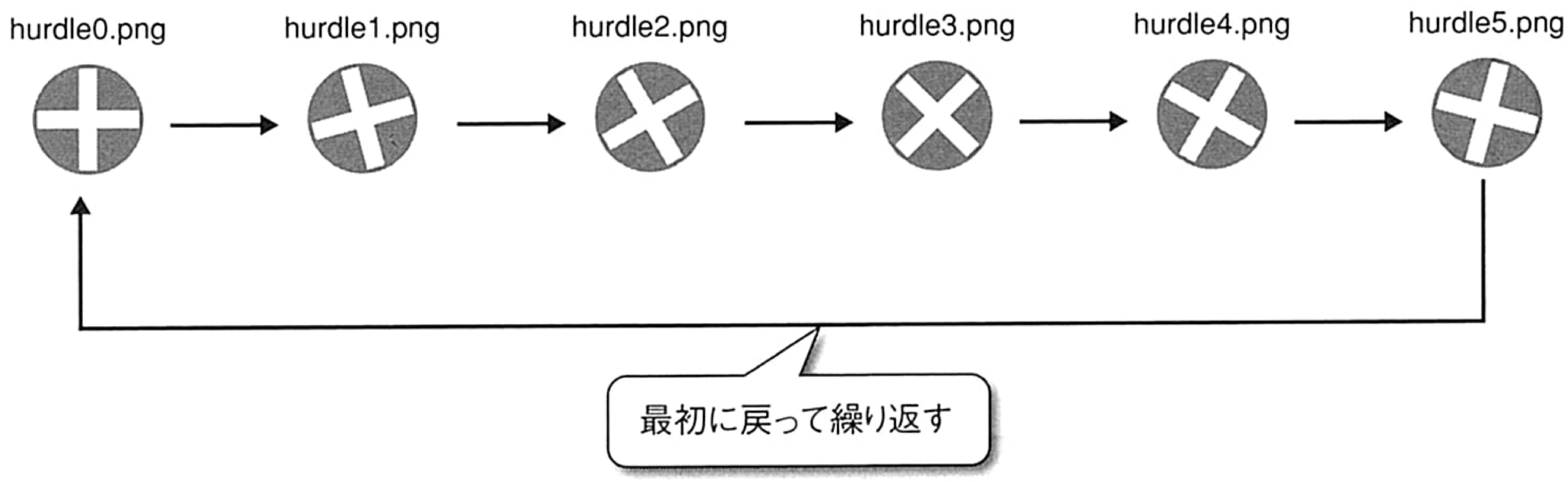
今度は障害物を動かすプログラムを作りましょう。まずは障害物を動かす方法について検討します。

障害物は画面の右から左に向かって転がってきます(Fig.19)。これは転がるアニメーション(Fig.20)を再生しながら、障害物の座標を変化させれば実現できます。画面の左端に達したら、右端から再出現するようにします。また、色々な速度の障害物があった方が面白いので、速度はランダムに決めることにしましょう。

▼ Fig.19: 障害物の移動

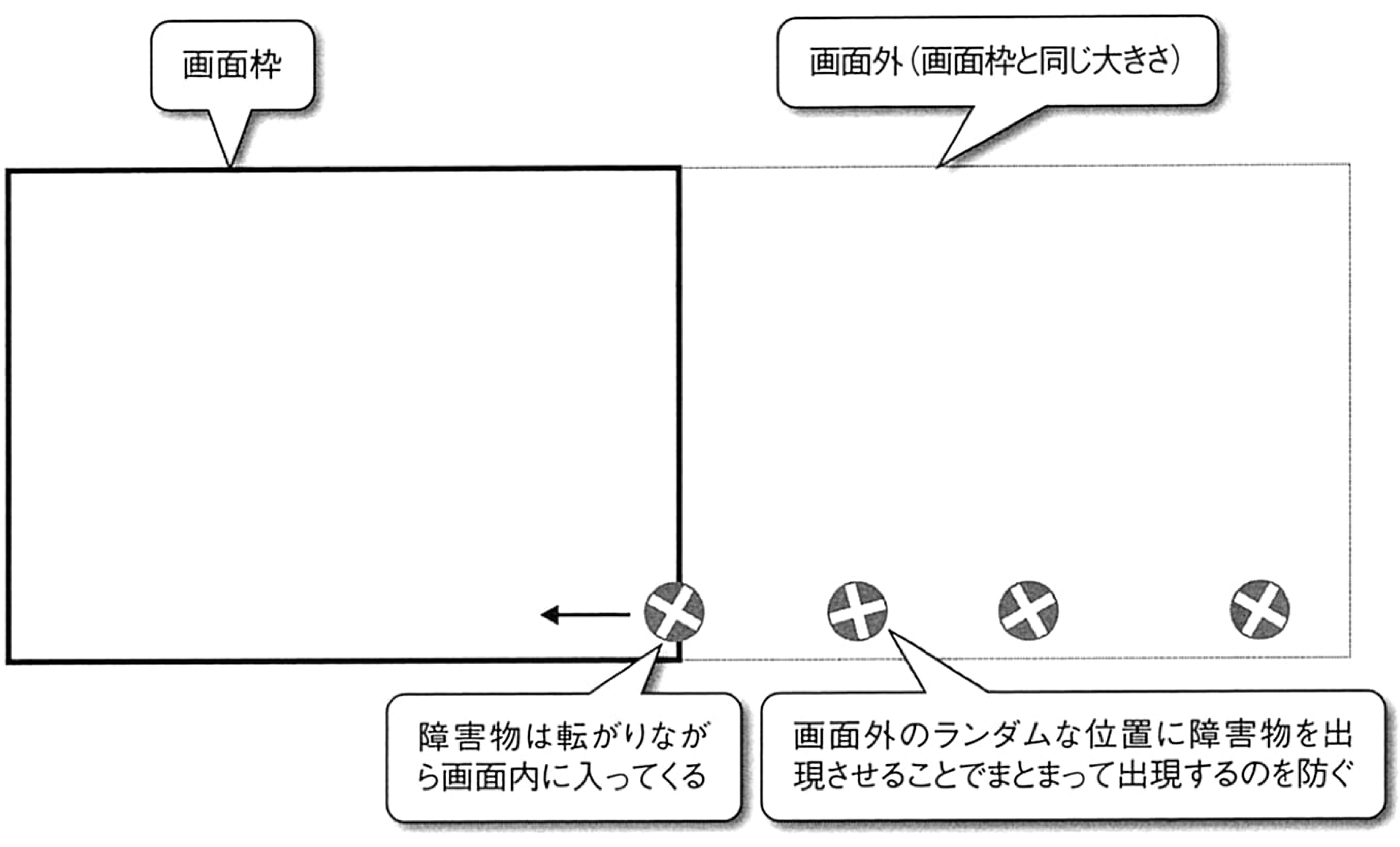


▼ Fig.20: 転がるアニメーション



画面上には複数の障害物を出現させます。最初に障害物を出現させるときには、障害物がまとまって出現しないための工夫が必要です。ここでは画面外のランダムな位置に配置することにしました(Fig.21)。

▼ Fig.21: 障害物の配置



障害物を動かすプログラム

障害物を動かすプログラムは、Hurdleオブジェクトとして定義しました(List4)。自機と同様に、スプライトを作成し、アニメーションや当たり判定を設定します。当たり判定は、

```
obj.sprite.initHit(hurdleHit);
```

のように初期化します。

hurdleHitは、

```
hurdleHit=new Array(0.25, 0.25, 0.75, 0.75);
```

のように初期化しました。自機と同じく、見た目よりも少し小さめの当たり判定です。

障害物の座標と速度は、

```
obj.sprite.setXY(randFloat2(XMAX+1, XMAX*2), YMAX);
obj.vx=-randFloat2(HURDLE_SPEED_MIN, HURDLE_SPEED_MAX);
```

のように初期化します。座標は画面外のランダムな位置です。randFloat2は指定した範囲の実数をランダムに生成します(p.77のList5)。

一定時間ごとに実行する更新処理(onUpdate関数)では、障害物の座標を変化させ、アニメーションを進行させます。また、

```
if (x<=-1) {
    x=XMAX+1;
    vx=-randFloat2(HURDLE_SPEED_MIN, HURDLE_SPEED_MAX);
}
```

のように座標を調べて、画面左端から障害物がはみ出したときには、画面右端に戻して再出現させます。このとき、速度もランダムに決め直します。

List4: 障害物を動かす(jumpKey.js)

```
// 障害物の最小スピード、最大スピード
var HURDLE_SPEED_MIN=0.1;
var HURDLE_SPEED_MAX=0.3;

// 障害物のオブジェクト
function Hurdle() {
  var obj=this;

  // スプライトの作成
  obj.sprite=game.createSprite();

  // アニメーションの設定
  obj.sprite.initAnim(hurdleAnim);

  // 当たり判定の設定
  obj.sprite.initHit(hurdleHit);
```

```
    // 座標と速度をランダムに決める
    obj.sprite.setXY(randFloat2(XMAX+1, XMAX*2), YMAX);
    obj.vx=-randFloat2(HURDLE_SPEED_MIN, HURDLE_SPEED_MAX);

    // ゲーム独自の更新処理
    obj.sprite.onUpdate=function() {

      // 座標と速度を取得する
      var x=obj.sprite.getX(), y=obj.sprite.getY();
      var vx=obj.vx;

      // 座標の更新
      x+=obj.vx;

      // 画面の左端に出たら、画面の右端に戻り、
      // 速度をランダムに決め直す
      if (x<=-1) {
        x=XMAX+1;
        vx=-randFloat2(HURDLE_SPEED_MIN, HURDLE_SPEED_MAX);
      }

      // スプライトの座標を更新する
      obj.sprite.setXY(x, y);

      // 速度のプロパティを更新する
      obj.vx=vx;

      // 転がるアニメーションを再生する
      obj.sprite.stepAnim();
    }
  }
```

キーボードで障害物を越えるゲームの本体

ゲーム本体のプログラムは、オブジェクトの作成や、アニメーションや当たり判定の初期化などを行います(List5)。ゲームの初期化を行うinit関数と、メインルーチンのmain関数を定義しました。

init関数では、画面上のスプライトを全て消去してから、自機と障害物のオブジェクトを作成します。スプライトの消去には、Gameオブジェクトのclear関数を使います(p.68のList3)。このclear関数では、

```
obj.divField.innerHTML="";
```

として、画面上にある全ての画像を削除します。innerHTMLプロパティは、指定した要素の内部に任意のHTMLドキュメントを書き込むために使います。ここではフィールド領域を表すdiv要素(divField)の内部を空にすることによって、div要素内に追加されたスプライト用のimg要素を全て削除しています。

　また、

```
obj.sprite=new Array();
```

のようにスプライトの配列を初期化することにより、Spriteオブジェクトも削除します。spriteプロパティを新しく作成した配列で上書きすることにより、古い配列は捨てられます。この処理はJavaScriptのガベージコレクション機能が行います。

List5: キーボードで障害物を越えるゲームの本体(jumpKey.js)

```
// 障害物の数
var HURDLE_COUNT=5;

// ゲーム本体、自機、障害物のオブジェクト
var game;
var player;
var hurdle;

// 自機、障害物、消滅のアニメーション
var playerWalkAnim;
var hurdleAnim;
var vanishAnim;

// 自機と障害物の当たり判定
var playerHit;
var hurdleHit;

// ゲームの開始
function init() {
  var i;

  // スプライトを全て消去する
  game.clear();

  // 自機の作成
  player=new Player();
```

```
  // 障害物の作成
  hurdle=new Array();
  for (i=0; i<HURDLE_COUNT; i++) {
    hurdle[i]=new Hurdle();
  }
}

// メインルーチン
function main() {

  // ゲームオブジェクトの作成
  game=new Game();

  // 自機が歩くアニメーション
  playerWalkAnim=new Array(
    "pcWalk0.png", "pcWalk2.png", "pcWalk0.png",
    "pcWalk1.png", "pcWalk3.png", "pcWalk1.png"
  );

  // 障害物が転がるアニメーション
  hurdleAnim=new Array(
    "hurdle0.png", "hurdle1.png", "hurdle2.png",
    "hurdle3.png", "hurdle4.png", "hurdle5.png"
  );

  // 自機が消滅するアニメーション
  vanishAnim=new Array(
    "void.png", "vanish8.png", "vanish6.png",
    "vanish4.png", "vanish2.png", "vanish0.png",
    "vanish1.png", "vanish2.png", "vanish3.png",
    "vanish4.png", "vanish5.png", "vanish6.png",
    "vanish7.png", "vanish8.png", "void.png"
  );

  // 自機と障害物の当たり判定
  playerHit=new Array(0.25, 0.25, 0.75, 0.75);
  hurdleHit=new Array(0.25, 0.25, 0.75, 0.75);

  // ゲームを開始する
  init();
}
```

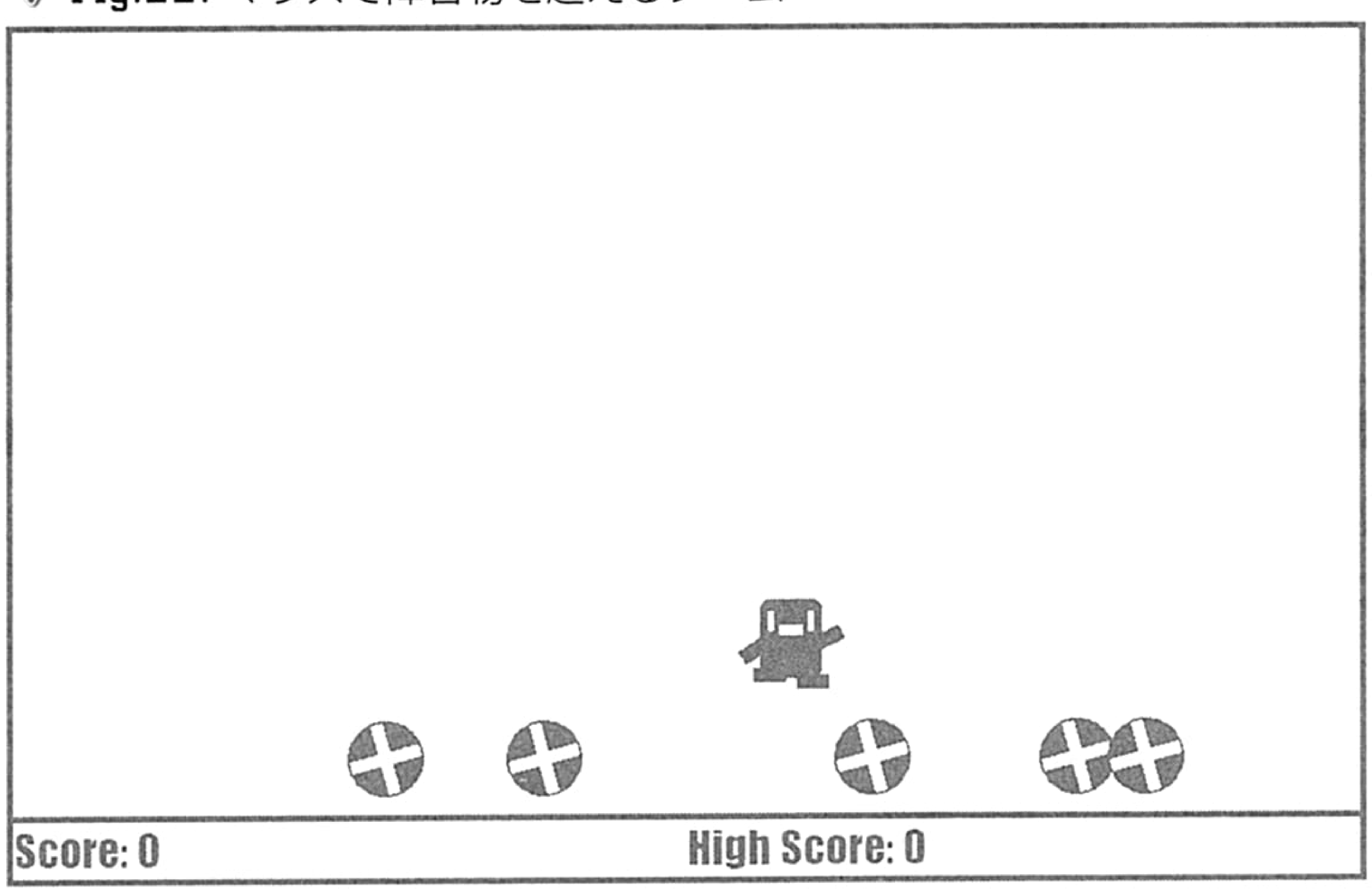
ガベージコレクション

ガベージコレクションとは、プログラムが使用したメモリ領域のうち、不要になった領域を自動的に解放して、再利用できるようにする機能です。JavaScriptやJavaはガベージコレクション機能を備えています。
ガベージコレクションの利点は、メモリの管理が簡単になるため、プログラムが記述しやすくなることです。欠点は、ガベージコレクションの処理には一般に時間がかかるため、速度が厳しく求められる用途には向かないことです。

マウスで障害物を越えるゲーム

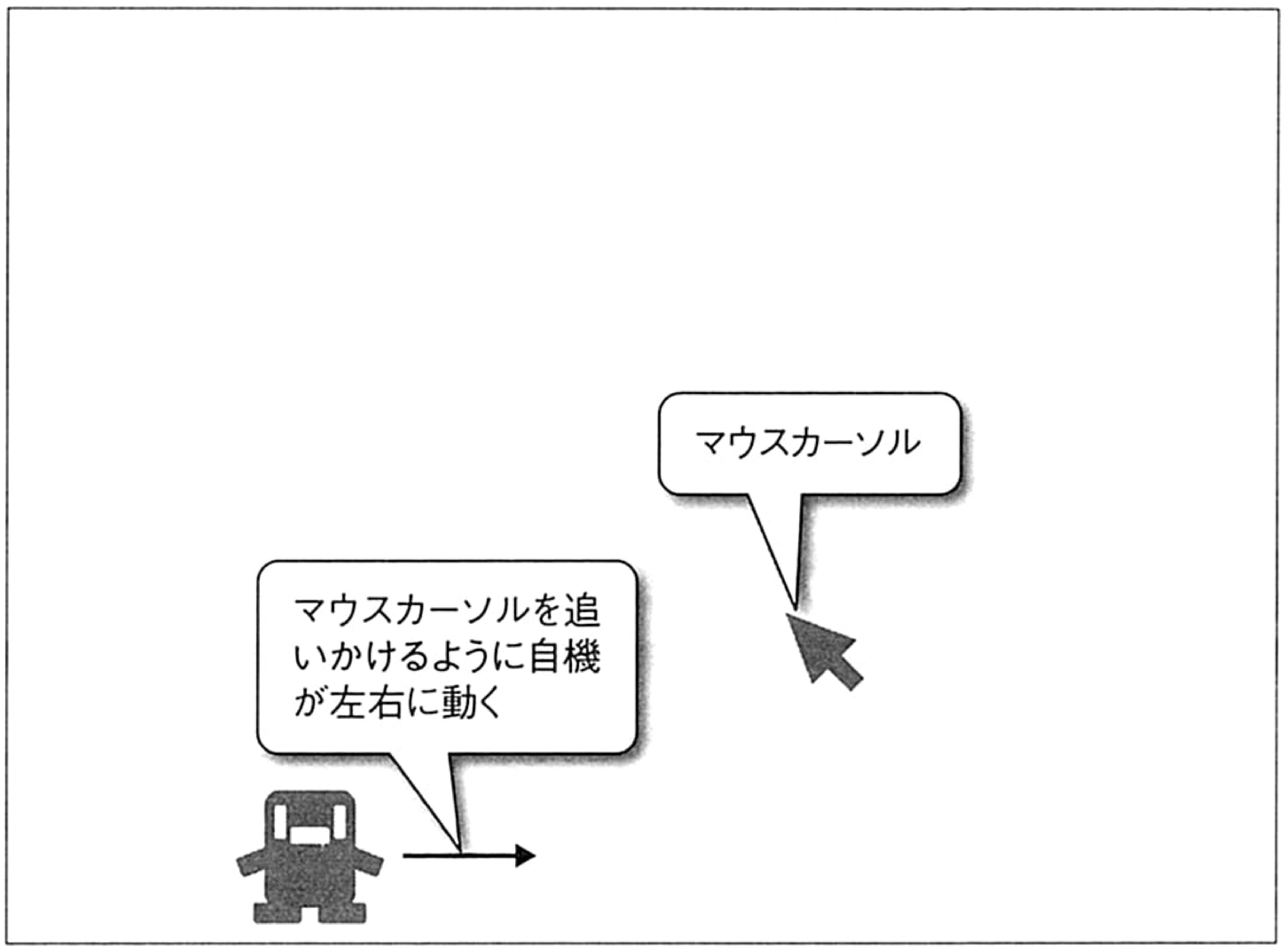
キーボードと同様に、マウスもゲームでよく利用される入力装置の1つです。そこで、障害物を越えるゲームを、マウスで操作するように改造してみました(Fig.22)。実行するにはjumpMouse.htmlをブラウザで開きます。自機の操作は全てマウスで行います。

▼ **Fig.22**: マウスで障害物を越えるゲーム



自機を移動させるには、マウスカーソルを動かします。自機はマウスカーソルを追いかけるように、左右に移動します(Fig.23)。

▼ Fig.23: 自機がマウスカーソルを追いかける



マウスの左ボタンを押すと、自機がジャンプします。ジャンプ中も自機を左右に移動させることができます。障害物に接触すると、自機は消滅します。

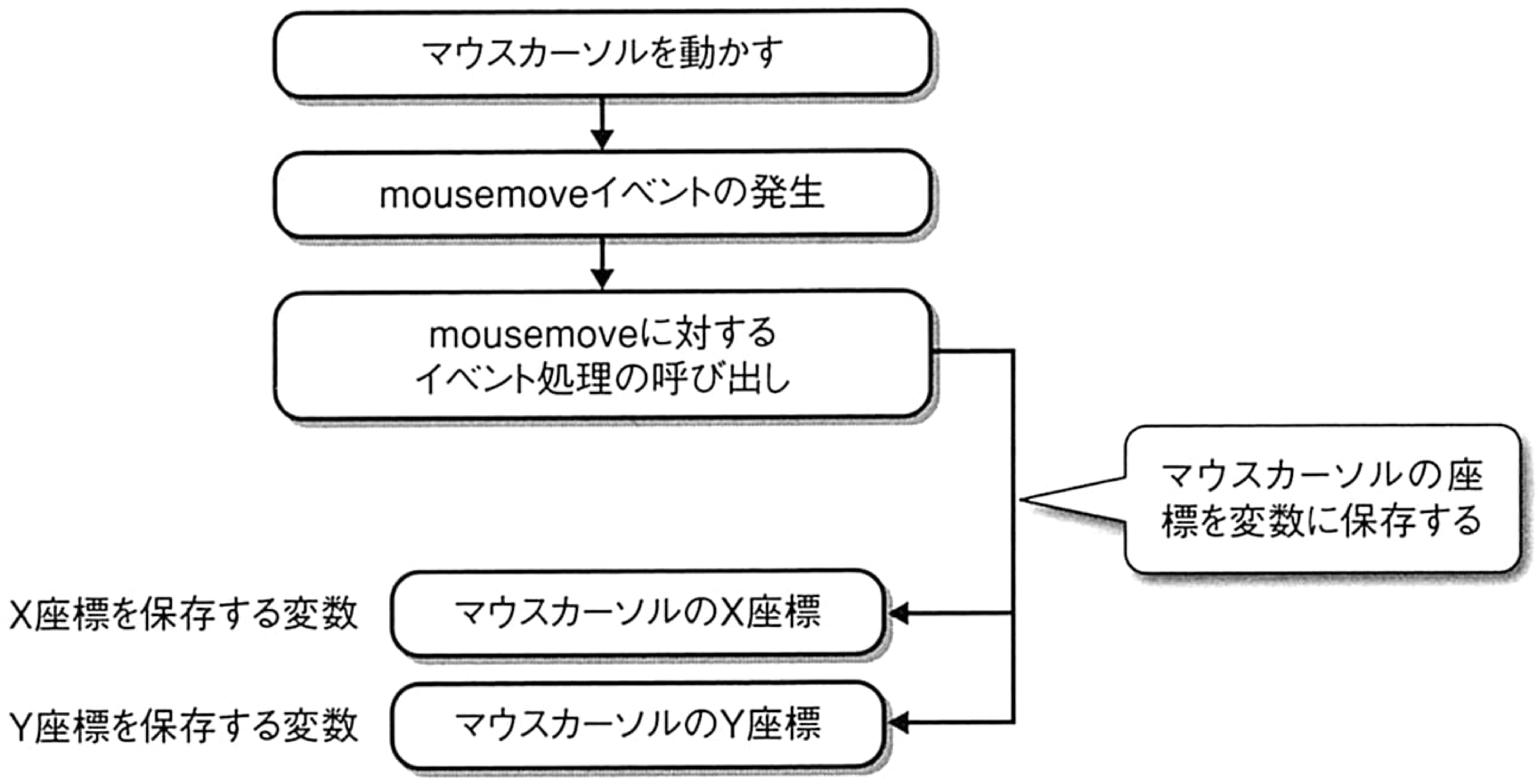
マウスの入力

マウスの入力を読み取るには、キーボードと同様にイベントを利用します。マウスの入力に関して、JavaScriptには次のようなイベントがあります。

- **mousemove**
  マウスカーソルを動かしたときに発生
- **mousedown**
  マウスボタンを押したときに発生
- **mouseup**
  マウスボタンを放したときに発生
- **mouseover**
  マウスがHTML要素の上に乗ったときに発生
- **mouseout**
  マウスがHTML要素の上から離れたときに発生

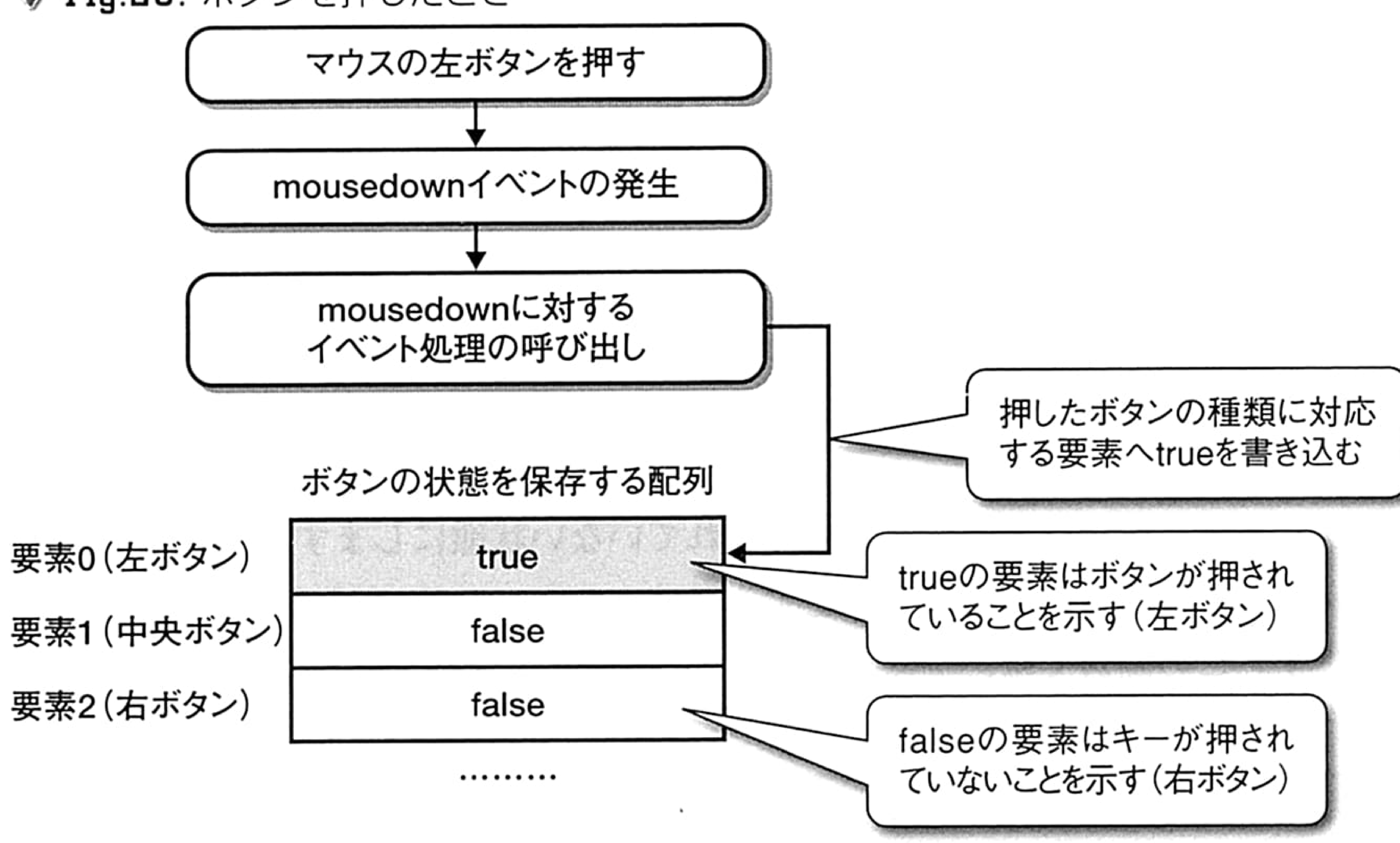
本書では、マウスカーソルの座標を得るためにmousemoveを使います。mousemoveイベントが発生したら、マウスカーソルの座標を変数に記録しておきます(Fig.24)。この変数を参照すれば、最新の座標を知ることができます。

Fig.24: マウスカーソルの座標を記録する

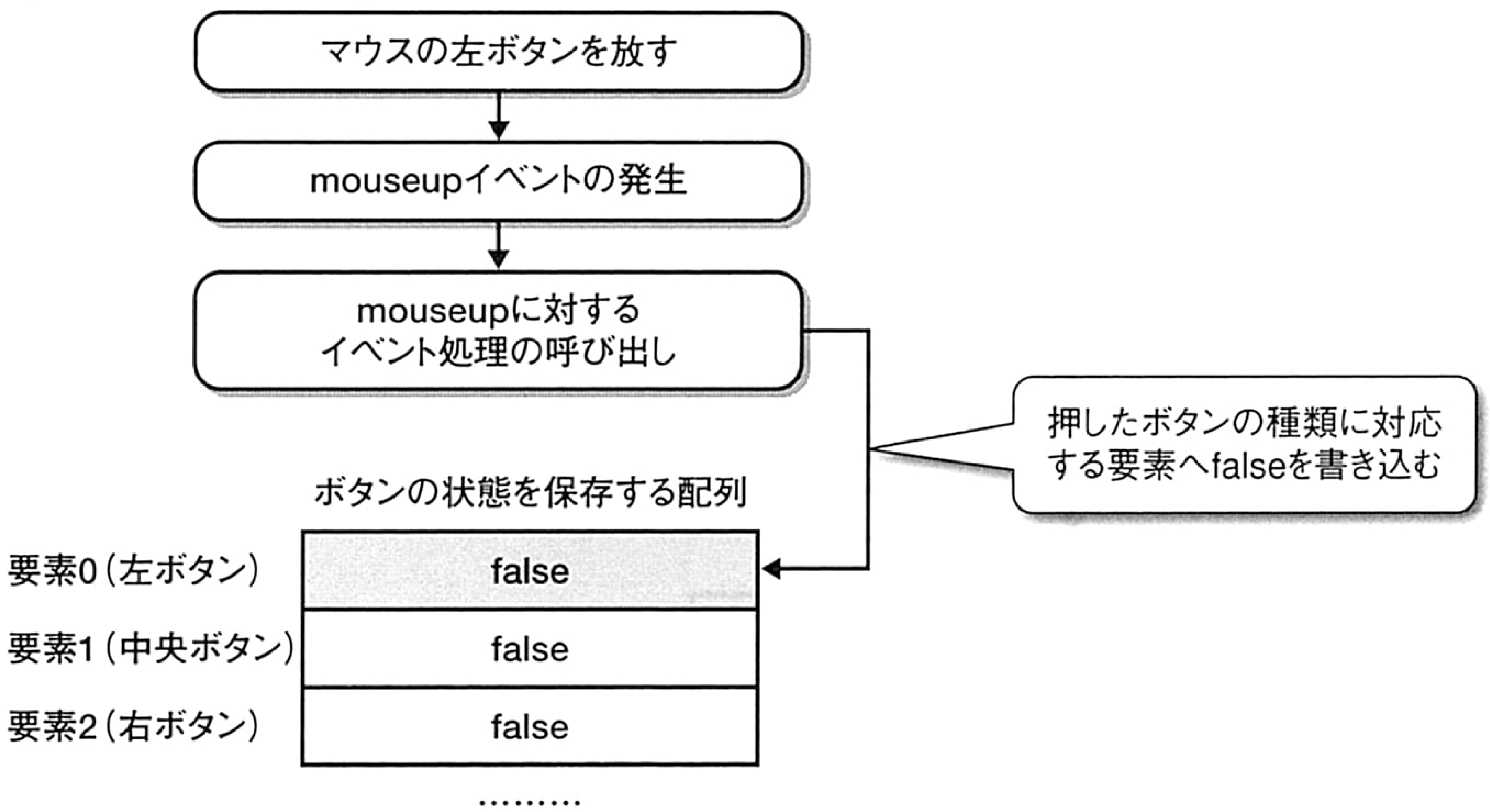


マウスボタンの状態を得るためには、mousedownとmouseupを使います。ボタンの状態を管理する配列を用意して、mousedownイベントが発生したら、ボタンの種類に対応する要素にtrueを書き込みます(Fig.25)。mouseupイベントが発生したら、要素にfalseを書き込みます(Fig.26)。この配列を調べれば、いつでもボタンの状態を知ることができます。

Fig.25: ボタンを押したとき



▼ Fig.26: ボタンを放したとき



マウスの入力を読み取るプログラム

マウスの入力を読み取る処理は、ライブラリ（common.js）のGameオブジェクトに追加します（List6）。まず、マウスカーソルの座標を保存する変数を、

```
obj.mouseX=0;
obj.mouseY=0;
```

のように定義します。マウスボタンの状態を表す配列は、

```
obj.mouseButton=new Array();
for (i=0; i<MOUSE_BUTTON_COUNT; i++) {
    obj.mouseButton[i]=false;
}
```

のように作成します。最初はどのボタンも押されていない状態にします。

mousemoveに対するイベント処理を記述するには、documentオブジェクトのonmousemove関数を定義します。この関数では、

```
document.onmousemove=function(e) {
    if (e) {
```

```
        obj.mouseX=(e.pageX-BORDER_SIZE)/CELL_SIZE+obj.viewX;
        obj.mouseY=(e.pageY-BORDER_SIZE)/CELL_SIZE+obj.viewY;
    } else {
        obj.mouseX=(event.x-BORDER_SIZE)/CELL_SIZE+obj.viewX;
        obj.mouseY=(event.y-BORDER_SIZE)/CELL_SIZE+obj.viewY;
    }
}
```

のように、引数eが有効な値かどうかによって、場合分けを行います。eはイベントオブジェクトです。eが有効な値ならば、pageXとpageYという2つのプロパティから、マウスカーソルの座標が得られます。eが無効な値ならば、eventオブジェクトのxプロパティとyプロパティを使います。

マウスカーソルの座標は、フィールド上の座標(キャラクターのサイズを1とする)に変換してから、mouseXプロパティとmouseYプロパティに格納します。こうすると、自機がマウスカーソルを追いかけるようなプログラムが簡単に作れます。

mousedownやmouseupに対するイベント処理を記述するには、documentオブジェクトのonmousedown関数やonmouseup関数を定義します。これらの関数でも、イベントオブジェクトを表す引数eの値が有効かどうかによって、場合分けを行います。

例えばonmousedown関数では、eの値が有効ならば、

```
obj.mouseButton[e.which-1]=true;
```

のような処理を行います。e.whichには、押したボタンに応じて、

- **左ボタン　:1**
- **中央ボタン:2**
- **右ボタン　:3**

のような値が入ります。一方、eの値が無効ならば、

```
if (event.button&1) obj.mouseButton[MOUSE_BUTTON_LEFT]=true;
if (event.button&4) obj.mouseButton[MOUSE_BUTTON_CENTER]=true;
if (event.button&2) obj.mouseButton[MOUSE_BUTTON_RIGHT]=true;
```

のように、eventオブジェクトのbuttonプロパティを使います。押したボタンの種類に応じて、buttonプロパティの特定のビットが1にセットされます。ボタンとビットの対応は、

- 左ボタン　：下から0ビット目
- 中央ボタン：下から2ビット目
- 右ボタン　：下から1ビット目

となっています(Fig.27)。プログラムではビット演算子の&を使って、ボタンの種類を調べています。

▼ **Fig.27**: ボタンとビットの対応



onmouseup関数でもonmousedown関数と同様の処理を行います。最後に、マウスカーソルの座標を返すgetMouseX関数とgetMouseY関数、ボタンの状態を返すisMouseButton関数を定義します。

▼ **List6**: マウスの入力を読み取る(common.js)

```
// マウスボタンを指定するための定数(左・中央・右)
var MOUSE_BUTTON_LEFT=0;
var MOUSE_BUTTON_CENTER=1;
var MOUSE_BUTTON_RIGHT=2;

// マウスボタンの数
var MOUSE_BUTTON_COUNT=3;

// ゲーム本体
function Game() {
  // ...(中略)...

  // マウスカーソルの座標を表すプロパティの初期化
  obj.mouseX=0;
  obj.mouseY=0;

  // マウスボタンの状態を表す配列の初期化
  obj.mouseButton=new Array();
  for (i=0; i<MOUSE_BUTTON_COUNT; i++) {

    // 最初はどのボタンも押されていない状態にする
    obj.mouseButton[i]=false;
  }

  // マウスカーソルを動かしたときのイベント処理
  document.onmousemove=function(e) {
```

```
  // 引数がある場合の処理(Firefoxなど)
  if (e) {
    obj.mouseX=(e.pageX-BORDER_SIZE)/CELL_SIZE+obj.viewX;
    obj.mouseY=(e.pageY-BORDER_SIZE)/CELL_SIZE+obj.viewY;
  } else

  // 引数がない場合の処理(Internet Explorerなど)
  {
    obj.mouseX=(event.x-BORDER_SIZE)/CELL_SIZE+obj.viewX;
    obj.mouseY=(event.y-BORDER_SIZE)/CELL_SIZE+obj.viewY;
  }
}

// マウスボタンを押したときのイベント処理
document.onmousedown=function(e) {

  // 引数がある場合の処理(Firefoxなど)
  if (e) {
    obj.mouseButton[e.which-1]=true;
  } else

  // 引数がない場合の処理(Internet Explorerなど)
  {
    if (event.button&1) obj.mouseButton[MOUSE_BUTTON_LEFT]=true;
    if (event.button&4) obj.mouseButton[MOUSE_BUTTON_CENTER]=true;
    if (event.button&2) obj.mouseButton[MOUSE_BUTTON_RIGHT]=true;
  }
}

// マウスボタンを放したときのイベント処理
document.onmouseup=function(e) {

  // 引数がある場合の処理(Firefoxなど)
  if (e) {
    obj.mouseButton[e.which-1]=false;
  } else

  // 引数がない場合の処理(Internet Explorerなど)
  {
    if (event.button&1) obj.mouseButton[MOUSE_BUTTON_LEFT]=false;
    if (event.button&4) obj.mouseButton[MOUSE_BUTTON_CENTER]=false;
    if (event.button&2) obj.mouseButton[MOUSE_BUTTON_RIGHT]=false;
  }
}

// マウスカーソルのX座標を返す
obj.getMouseX=function() {
```

```
    return obj.mouseX;
  }

  // マウスカーソルのY座標を返す
  obj.getMouseY=function() {
    return obj.mouseY;
  }

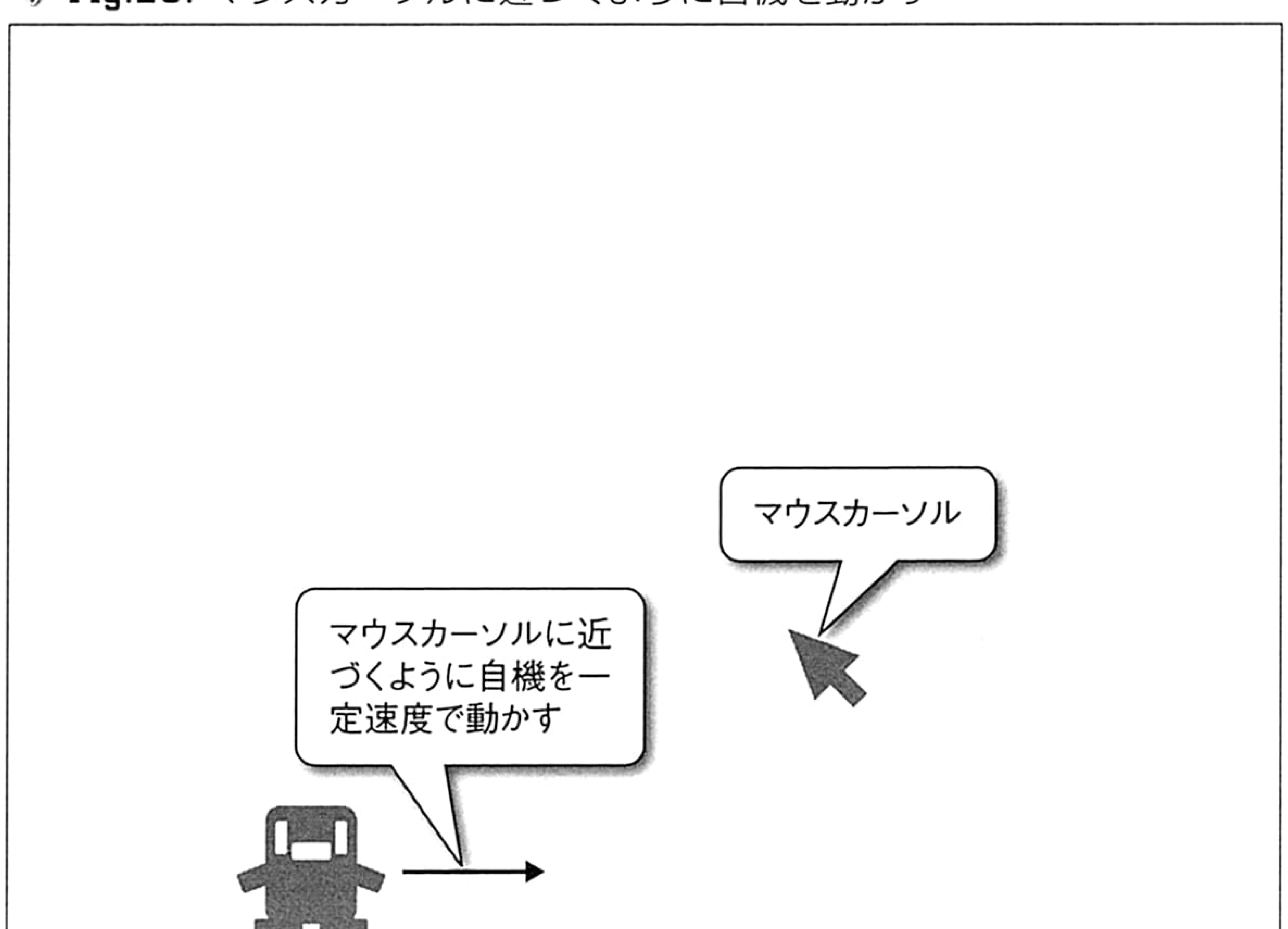
  // 指定したマウスボタンが押されているときにtrueを返す
  obj.isMouseButton=function(index) {
    return obj.mouseButton[index];
  }
}
```

マウスで自機を動かす

マウスの入力処理ができたところで、マウスを使って自機を動かす方法を詳しく検討しましょう。マウスの左ボタンを押したときに自機をジャンプさせるのは、スペースキーを押したときにジャンプさせるのとほとんど同じなので、特に難しいことはありません。

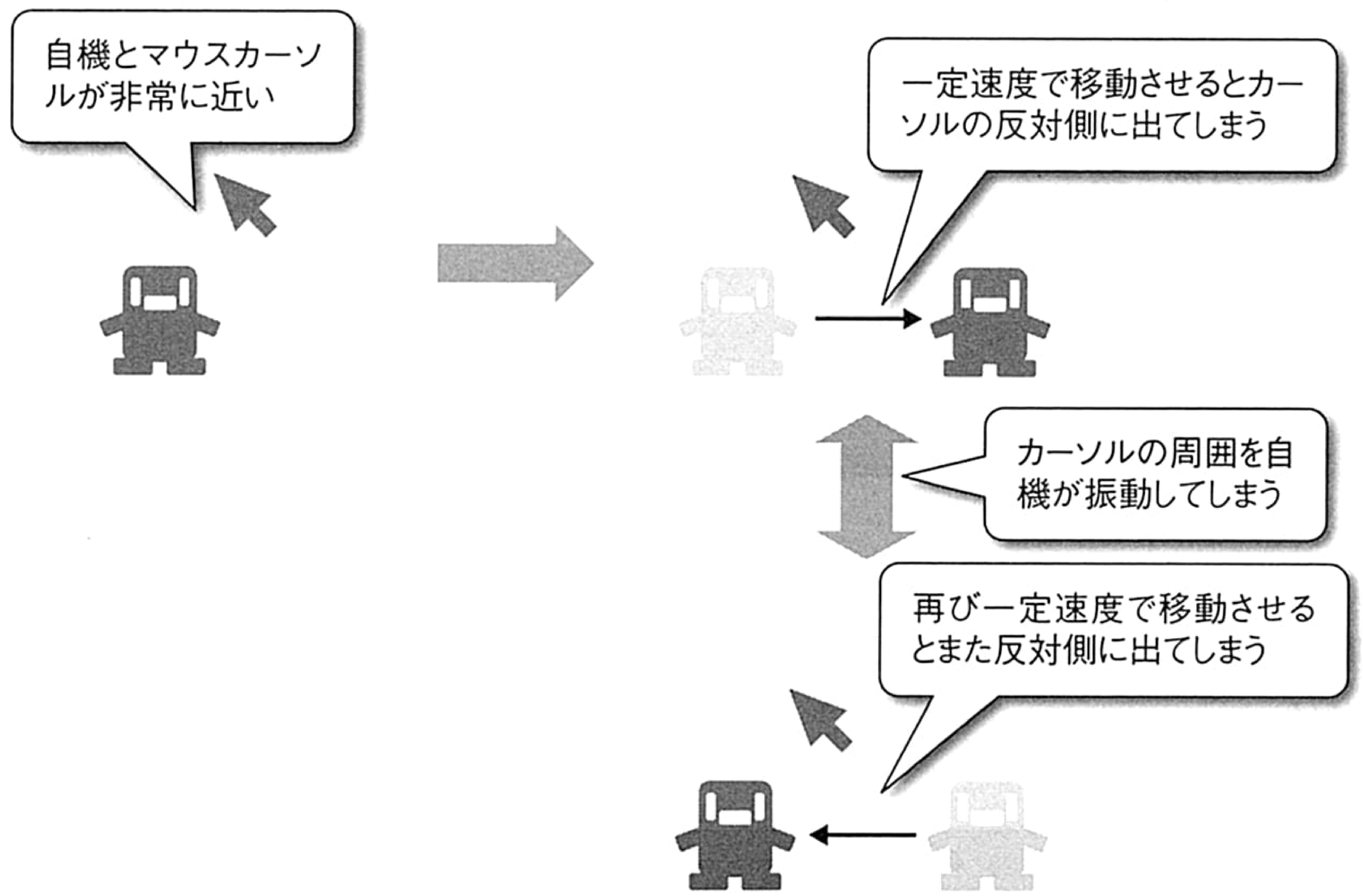
少し考える必要があるのは、マウスカーソルを追いかけるように自機を移動させる方法です。それには、マウスカーソルの座標と自機の座標を比べます(Fig.28)。マウスカーソルが左にあれば自機を左へ、右にあれば自機を右へ、決まった速度で動かします。

Fig.28: マウスカーソルに近づくように自機を動かす

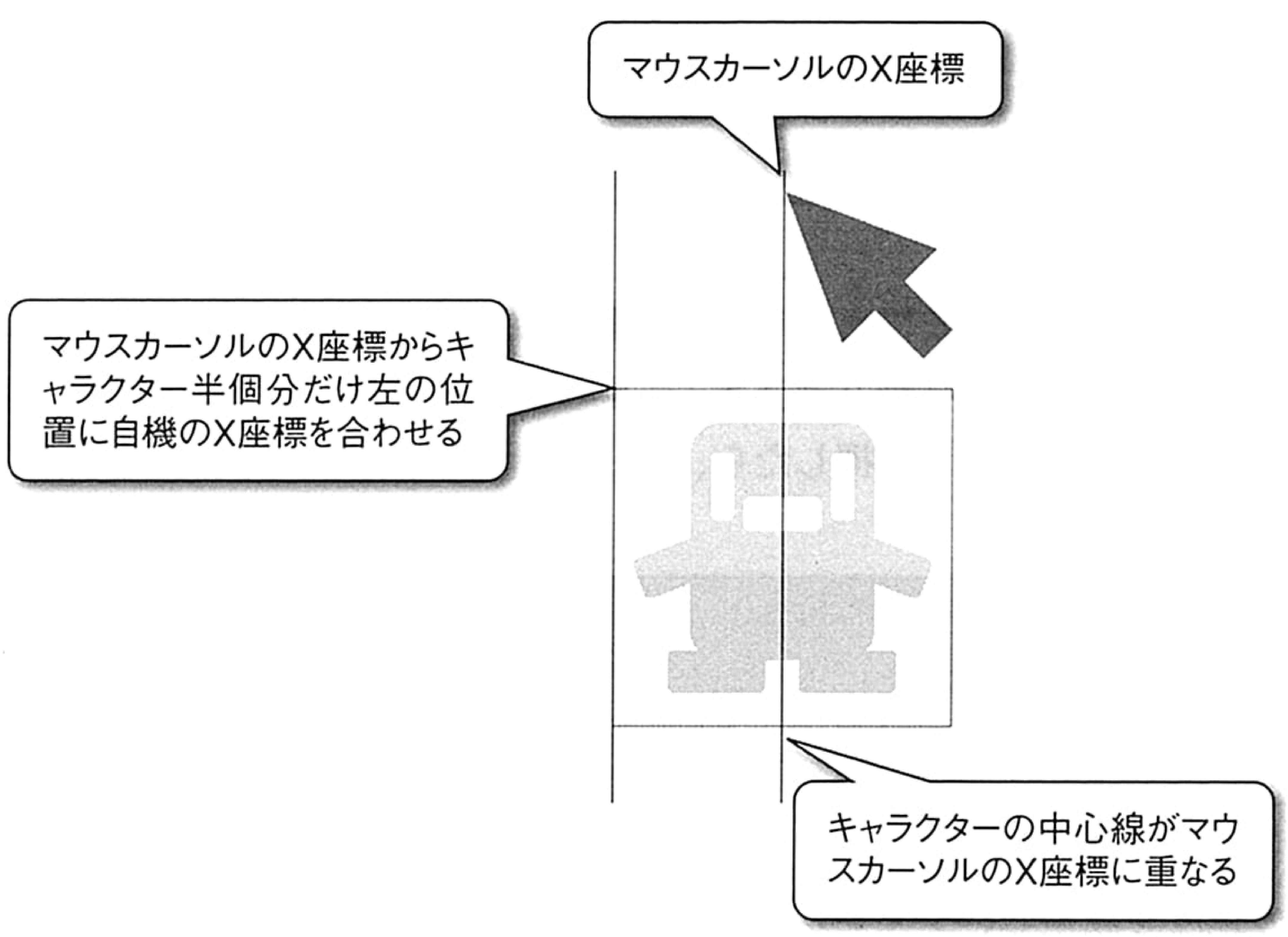


マウスカーソルと自機が非常に近いときに、自機を決まった速度で動かすと、マウスカーソルの周りを自機が振動してしまうことがあります(Fig.29)。そこで、非常に近いときには、決まった速度で動かすのではなく、自機をマウスカーソルと同じ座標に移動させます(Fig.30)。

▼ Fig.29: マウスカーソルと自機が非常に近い場合



▼ Fig.30: 自機をマウスカーソルの座標に移動させる



自機のX座標(スプライトの左端の座標)は、マウスカーソルのX座標よりも自機半個分だけ左にするとよいでしょう。自機の中心線がマウスカーソルのX座標と重なるように動くので、操作しやすくなります。

マウスで自機を動かすプログラム

List7はマウスで自機を動かすプログラムです。自機を表すPlayerオブジェクトを定義しました。キーボードで自機を動かすプログラム(p.104のList3)との違いは、alive関数(通常状態の処理)だけなので、この関数を抜粋して説明します。

最初に、

```
var mx=game.getMouseX()-0.5;
```

のように、マウスカーソルの座標を取得し、自機の移動先を計算します。移動先は、マウスカーソルの座標よりも自機半個分だけ左にします。

自機の移動は、

```
if (Math.abs(mx-x)<=PLAYER_VX) {
    x=mx;
} else {
    if (mx<x) x-=PLAYER_VX; else x+=PLAYER_VX;
}
```

のように行います。Math.abs(mx－x)は移動先と自機の距離です。Mathオブジェクトのabs関数は、引数の絶対値を返します。

距離が非常に近いときには、移動先に自機を直接移動させます。距離が遠いときには、移動先へ近づく方向に自機を一定速度で移動させます。

地上にいるときにマウスの左ボタンを押したら、

```
if (game.isMouseButton(MOUSE_BUTTON_LEFT)) {
    vy=PLAYER_VY;
    obj.jump=true;
}
```

のように、上方向の速度を設定し、ジャンプ状態にします。キーボードの代わりにマウスを使いますが、ジャンプ自体の処理はキーボードの場合と全く同じです。

List7: マウスで自機を動かす(jumpMouse.js)

```
// 自機のオブジェクト
function Player() {
  // ...(中略)...

  // 通常状態の処理
  obj.alive=function() {

    // 座標と速度を取得する
    var x=obj.sprite.getX(), y=obj.sprite.getY();
    var vy=obj.vy;

    // マウスカーソルの座標から、
    // 自機の移動先の座標を計算する
    var mx=game.getMouseX()-0.5;

    // 移動先と自機が近いときには、
    // 自機を移動先まで動かす
    if (Math.abs(mx-x)<=PLAYER_VX) {
      x=mx;
    } else

    // 移動先と自機が遠いときには、
    // 自機を一定の速度で移動先に近づくように動かす
    {
      if (mx<x) x-=PLAYER_VX; else x+=PLAYER_VX;
    }

    // 画面からはみ出したら座標を修正する
    if (x<0) x=0;
    if (x>XMAX) x=XMAX;

    // ジャンプしていないときの処理
    if (!obj.jump) {

      // マウスの左ボタンでジャンプする
      if (game.isMouseButton(MOUSE_BUTTON_LEFT)) {
        vy=PLAYER_VY;
        obj.jump=true;
      }

      // 左右に移動したら、
      // 歩くアニメーションを再生する
      if (x!=obj.sprite.getX()) {
        obj.sprite.stepAnim();
      }
    } else
```

```
    // ジャンプしているときの処理
    {
      // 座標と速度を更新する
      y+=vy;
      vy+=PLAYER_AY;

      // 画面の下端に達したら着地する
      if (y>=YMAX) {
        y=YMAX;
        obj.jump=false;
      }
    }

    // スプライトの座標を更新する
    obj.sprite.setXY(x, y);

    // 速度のプロパティを更新する
    obj.vy=vy;

    // 障害物に接触したときの処理
    if (game.testHitSprite(obj.sprite)) {

      // 消滅のアニメーションに切り替える
      obj.sprite.initAnim(vanishAnim);

      // 消滅状態にする
      obj.state=STATE_DEAD;
    }
  }
}
```

Chapter 4

スコア

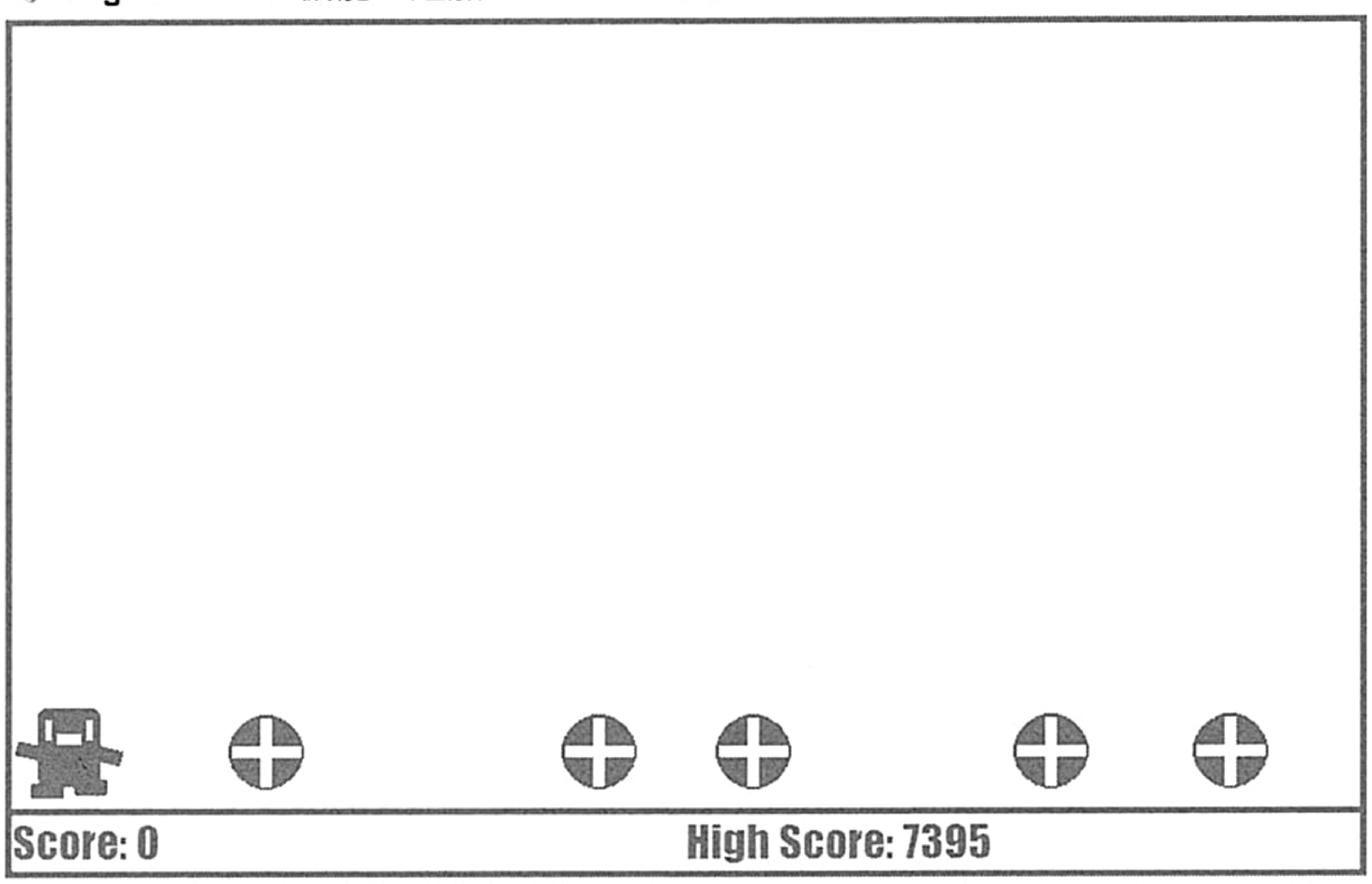
ゲームにスコア機能を付ける

多くのゲームにはスコア(得点)があります。スコアはゲームを上手にプレイできたかどうかの指標になるので、スコアがあるとより楽しくゲームを遊ぶことができます。

第3章で制作したジャンプゲームにスコア機能を追加してみました(Fig.1)。スコアは左下に、ハイスコア(スコアの最高記録)は右下に表示されます。

ゲームが始まるたびにスコアは0になりますが、ハイスコアは残ります。ブラウザでページを更新したときには、ハイスコアも0に戻ります。

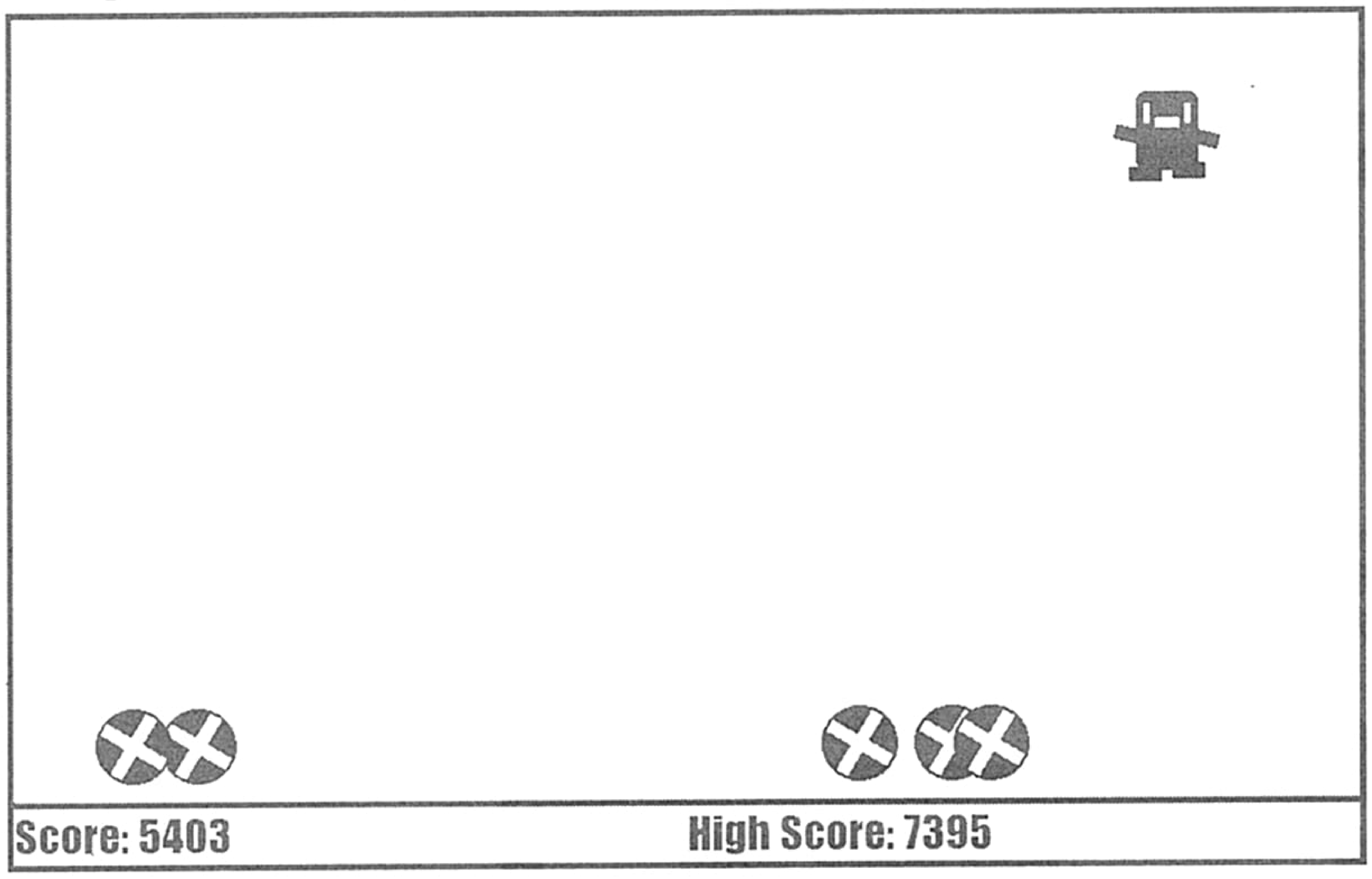
▼ **Fig.1**: スコア機能を追加したジャンプゲーム



このゲームでは障害物を避けて長く生き残るほど、高いスコアが得られるようにしました(Fig.2)。また、自機が画面の右にいるほど、スコアが速く上がるようにしました。

障害物は画面の右端から出現するので、右にいるほど避けるのが難しくなります。左にいれば簡単に避けられますが、高いスコアを得るには右に行かなければならないということで、ゲームのスリルを増すことを狙っています。

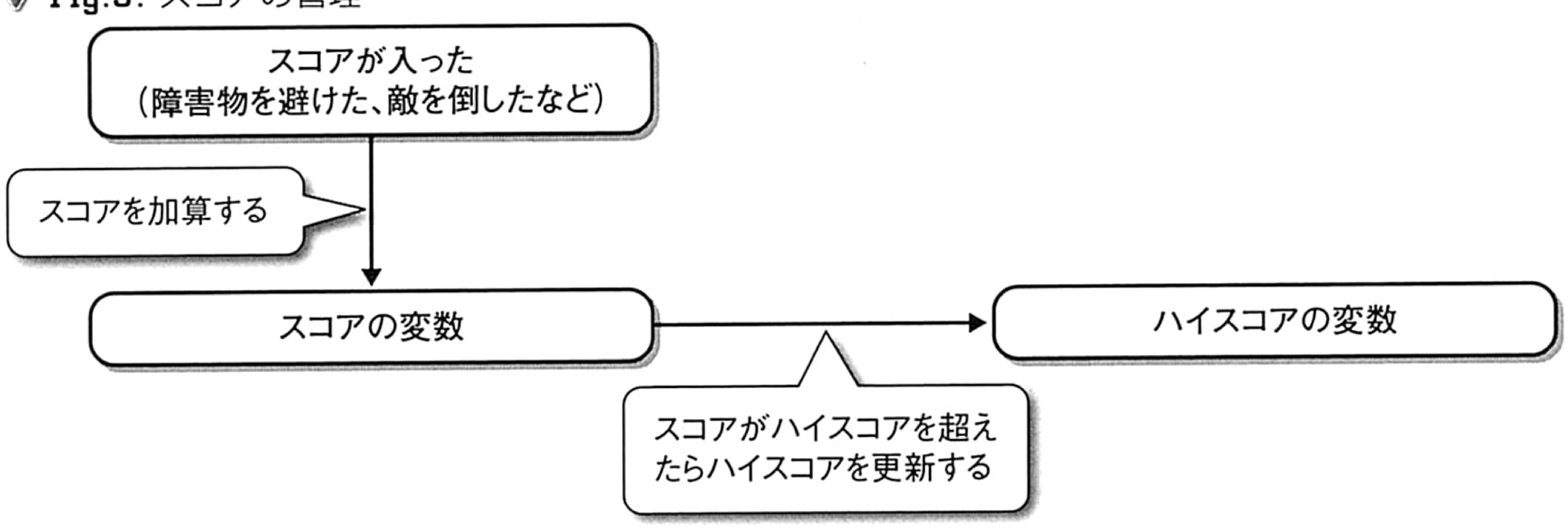
▼ Fig.2: 生き残るほど高いスコアが得られる



スコアの管理と表示

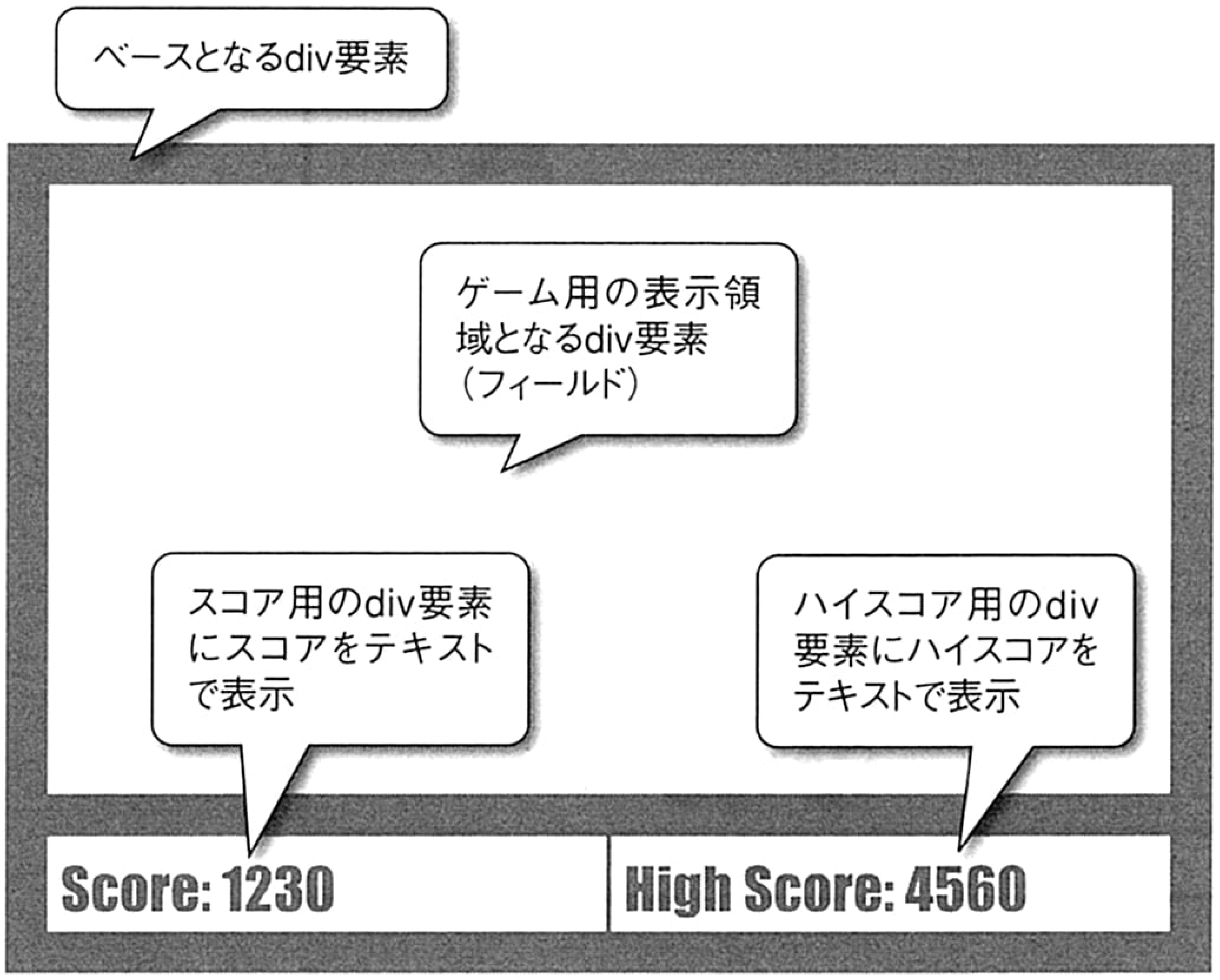
スコアを管理するには、スコアとハイスコアを保持する変数を用意します(Fig.3)。ゲームで障害物を避けたり、敵を倒したりしたら、スコアを加算します。スコアを加算したときに、スコアがハイスコアを上回ったら、スコアを新しいハイスコアにします。

▼ Fig.3: スコアの管理



スコアを表示する方法はいくつかありますが、本書ではHTMLのdiv要素を使います。第2章でゲーム画面を作ったときに、スコアとハイスコアの表示領域を用意しました(Fig.4)。ここにスコアとハイスコアをテキストで表示します。

▼ Fig.4: スコアの表示



スコアを管理するプログラム

List1はスコアを管理するプログラムです。ライブラリ(common.js)から、スコアに関する部分を抜粋しました。

スコア表示用のdiv要素は、

```
obj.divScore=document.createElement("div");
obj.divBase.appendChild(obj.divScore);
```

のように作ります。ハイスコア表示用のdiv要素も同様です。

次に、div要素のstyleプロパティを使って、位置・サイズ・色・フォントなどを設定します。フォントについては、

```
style.fontFamily="Impact, sans-serif";
style.fontSize="24pt";
```

のように設定しました(List1では定数を使っています)。"Impact, sans-serif"というのは、Impactフォントがあればそれを使い、なければsans-serifフォント(サンセリフ：文字の線の端に飾りがない字体)の中から適当なものを使うための設定です。"24pt"というのは、24ポイントのサイズを表します。

スコアはscoreプロパティに、ハイスコアはhighScoreプロパティに記録します。そして、スコアとハイスコアを操作するために、次のような関数を用意します。

- getHighScore: ハイスコアの取得
- setHighScore: ハイスコアの設定
- getScore: スコアの取得
- setScore: スコアの設定
- addScore: スコアの加算

例えばスコアを設定するsetScore関数では、

```
obj.score=score;
```

のように、引数scoreをscoreプロパティに代入します。そして、

```
if (obj.score>MAX_SCORE) {
    obj.score=MAX_SCORE;
}
```

のように、スコアが最大値(999999999)を超えないようにします。スコアに最大値を設けるのは、表示できる値の桁数に制限があるためと、プログラムで扱える値の大きさにも限界があるからです。

次に、

```
obj.divScore.innerHTML="Score: "+obj.score;
```

のように、div要素にスコアを表示します。スコアの値の前に"Score: "を追加することによって、

```
Score: 0
```

のような文字列にします。

最後に、

```
if (obj.score>obj.highScore) {
    obj.setHighScore(obj.score);
}
```

として、スコアがハイスコアが上回ったときには、スコアを新たなハイスコアに設定します。
　スコアを加算するaddScore関数は、例えば、

```
game.addScore(100);
```

のように使うと、現在のスコアに100が加算されます。これは、

```
game.setScore(game.getScore()+100);
```

と書いてもよいのですが、ゲームではスコアを加算する処理をよく行うので、簡単のためにaddScore関数を用意しました。

List1: スコアの管理(common.js)

```
// スコア表示領域の高さ
var SCORE_HEIGHT=40;

// スコア表示用のフォントの種類とサイズ
var SCORE_FONT_FAMILY="Impact, sans-serif";
var SCORE_FONT_SIZE="24pt";

// スコアの最大値
var MAX_SCORE=999999999;

// ゲーム本体
function Game() {
  // ...(中略)...

  // スコア表示領域
  obj.divScore=document.createElement("div");
  obj.divBase.appendChild(obj.divScore);
  style=obj.divScore.style;
  style.overflow="hidden";
  style.position="absolute";
  style.backgroundColor=FIELD_COLOR;
  style.left=BORDER_SIZE+"px";
  style.top=FIELD_HEIGHT+BORDER_SIZE*2+"px";
  style.width=FIELD_WIDTH/2+"px";
  style.height=SCORE_HEIGHT+"px";
  style.fontFamily=SCORE_FONT_FAMILY;
  style.fontSize=SCORE_FONT_SIZE;

  // ハイスコア表示領域
  obj.divHighScore=document.createElement("div");
```

```
obj.divBase.appendChild(obj.divHighScore);
style=obj.divHighScore.style;
style.overflow="hidden";
style.position="absolute";
style.backgroundColor=FIELD_COLOR;
style.left=BORDER_SIZE+FIELD_WIDTH/2+"px";
style.top=FIELD_HEIGHT+BORDER_SIZE*2+"px";
style.width=FIELD_WIDTH/2+"px";
style.height=SCORE_HEIGHT+"px";
style.fontFamily=SCORE_FONT_FAMILY;
style.fontSize=SCORE_FONT_SIZE;

// ハイスコアの取得
obj.getHighScore=function() {
  return obj.score;
}

// ハイスコアの設定
obj.setHighScore=function(score) {

  // ハイスコアのプロパティを更新する
  obj.highScore=score;

  // スコアの最大値を超えないようにする
  if (obj.highScore>MAX_SCORE) {
    obj.highScore=MAX_SCORE;
  }

  // ハイスコアの表示
  obj.divHighScore.innerHTML="High Score: "+obj.highScore;
}

// 初期状態ではハイスコアを0に設定する
obj.setHighScore(0);

// スコアの取得
obj.getScore=function() {
  return obj.score;
}

// スコアの設定
obj.setScore=function(score) {

  // スコアのプロパティを更新する
  obj.score=score;

  // スコアの最大値を超えないようにする
```

```
    if (obj.score>MAX_SCORE) {
      obj.score=MAX_SCORE;
    }

    // スコアの表示
    obj.divScore.innerHTML="Score: "+obj.score;

    // スコアがハイスコアを上回っていたら
    // ハイスコアを設定する
    if (obj.score>obj.highScore) {
      obj.setHighScore(obj.score);
    }
  }

  // スコアの加算
  obj.addScore=function(score) {
    obj.setScore(obj.score+score);
  }

  // 初期状態ではスコアを0に設定する
  obj.setScore(0);
}
```

スコア機能を付けたジャンプゲーム

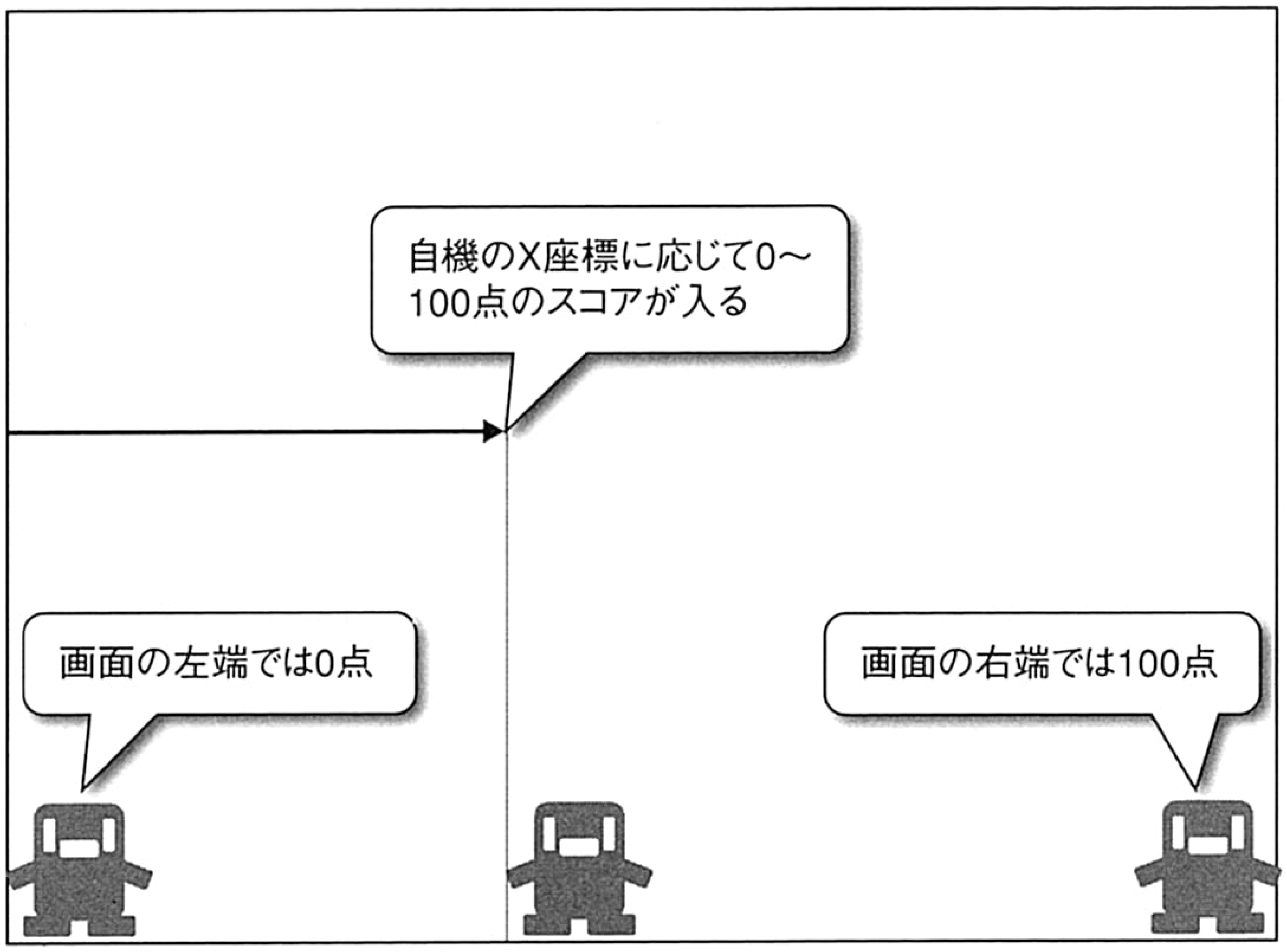
List1を使うと、第3章のゲームに対して簡単にスコア機能を追加することができます(List2)。Playerオブジェクトとinit関数を抜粋しましたが、内容は元のプログラム(p.104のList3)とほとんど同じで、異なるのはたった2行だけです。

Playerオブジェクトのalive関数では、

```
game.addScore(Math.floor(x/XMAX*100));
```

のようにスコアを加算します。障害物を避け続けている限り、スコアが得られます。自機のX座標(変数x)を使うことにより、自機が画面の右にいるほど高いスコアが得られるようにしました(Fig.5)。画面の左端にいると0点で、右にいくほどスコアが高くなり、右端では最大で100点になります。

▼ **Fig.5**: スコアの計算



init関数はゲームの初期化を行います。ここではゲームを始めるたびに、

```
game.setScore(0);
```

として、スコアを0にします。ハイスコアは0にはせずに、元の記録を残します。

▼ **List2**: スコア機能を付けたジャンプゲーム(jumpScore.js)

```
function Player() {
  // ...(中略)...

  // 通常状態の処理
  obj.alive=function() {
    var x=obj.sprite.getX(), y=obj.sprite.getY();
    var vy=obj.vy;

    // カーソルキーで左右に移動する
    if (game.isKey(KEY_LEFT)) x-=PLAYER_VX;
    if (game.isKey(KEY_RIGHT)) x+=PLAYER_VX;
    if (x<0) x=0;
    if (x>XMAX) x=XMAX;

    // ジャンプの処理
    if (!obj.jump) {
      if (game.isKey(KEY_SPACE)) {
        vy=PLAYER_VY;
        obj.jump=true;
      }
```

```
      if (x!=obj.sprite.getX()) {
        obj.sprite.stepAnim();
      }
    } else {
      y+=vy;
      vy+=PLAYER_AY;
      if (y>=YMAX) {
        y=YMAX;
        obj.jump=false;
      }
    }

    // 座標と速度の更新
    obj.sprite.setXY(x, y);
    obj.vy=vy;

    // 障害物に接触したときの処理
    if (game.testHitSprite(obj.sprite)) {
      obj.sprite.initAnim(vanishAnim);
      obj.state=STATE_DEAD;
    }

    // スコアの加算
    game.addScore(Math.floor(x/XMAX*100));
  }
}


// ゲームの開始
function init() {
  var i;

  // スプライトの消去
  game.clear();

  // スコアを0にする
  game.setScore(0);

  // 自機の作成
  player=new Player();

  // 障害物の作成
  hurdle=new Array();
  for (i=0; i<HURDLE_COUNT; i++) {
    hurdle[i]=new Hurdle();
  }
}
```

Chapter 5

敵

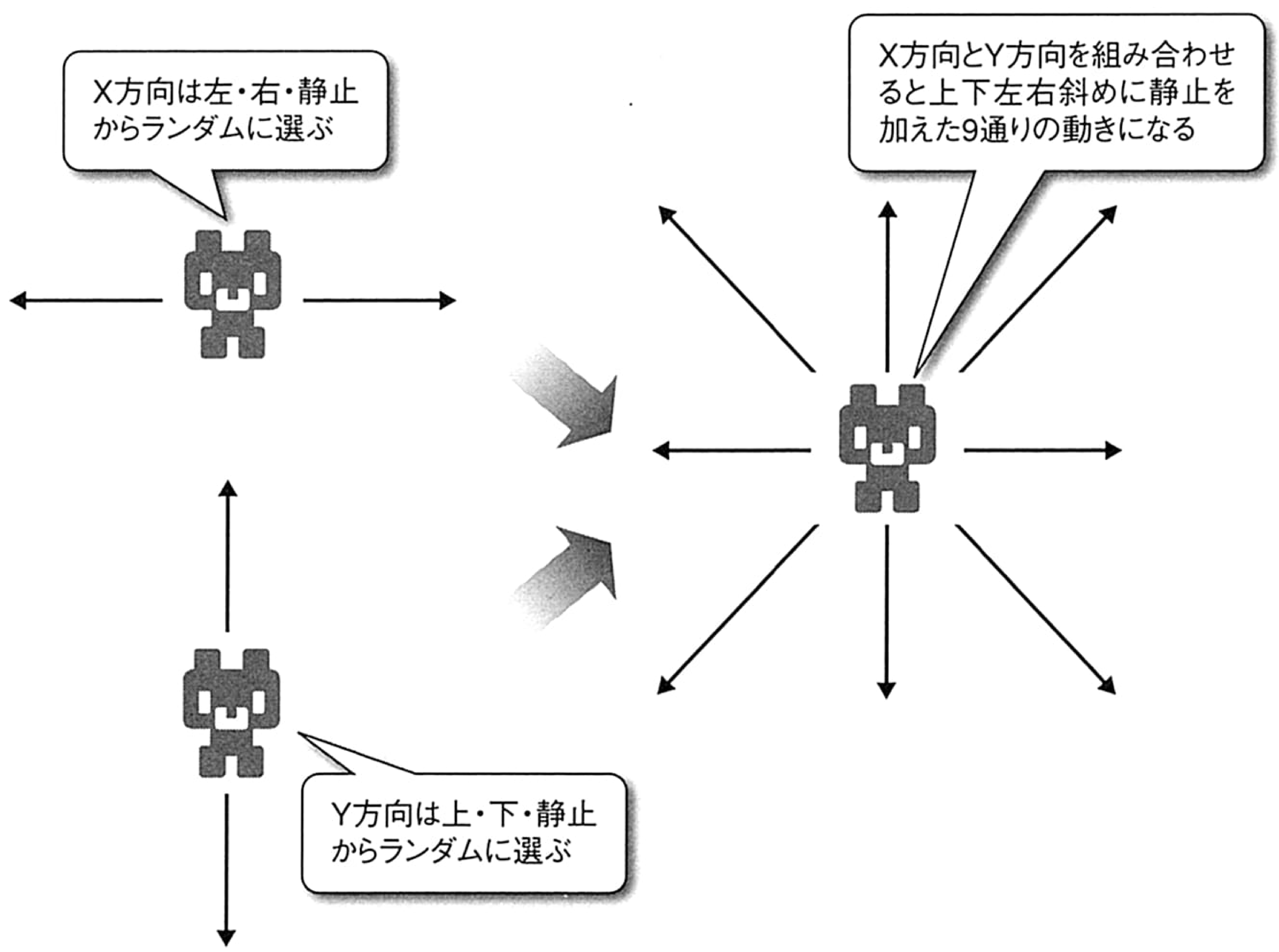
ランダムに動く敵

第3章・第4章では、障害物をジャンプして避けるゲームを作りました。この障害物は敵の一種といえますが、右から左に転がってくるだけです。多くのゲームでは、敵はもっと複雑な動きをします。そこで本章では、ランダムに動く敵と、自機を追いかける敵の作り方を解説します。

まず敵をランダムに動かす方法です。ランダムといっても、敵の速度をランダムに変え続けるだけでは、同じような場所を振動するような動きになってしまい、あまり面白くありません。そこで、ランダムでいて、なおかつ気の利いた動きをする敵を考えてみました。

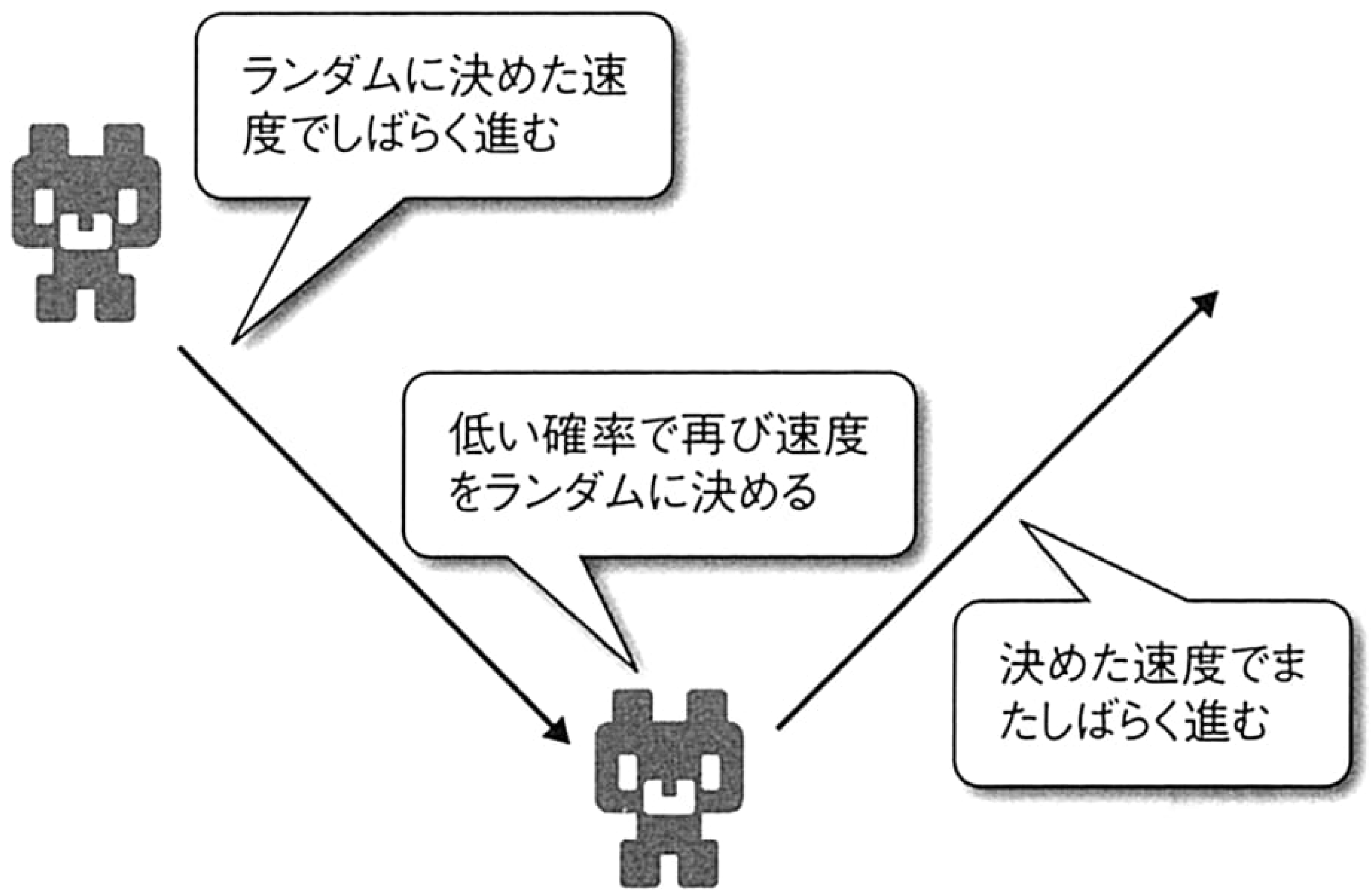
敵の速度はランダムに決めます(Fig.1)。X方向の速度は左・右・静止から、Y方向の速度は上・下・静止から、それぞれランダムに選びます。これで敵は、上下左右斜めの8方向に静止を加えた、9通りの動きをします。

▼ **Fig.1**: 敵の速度を決める



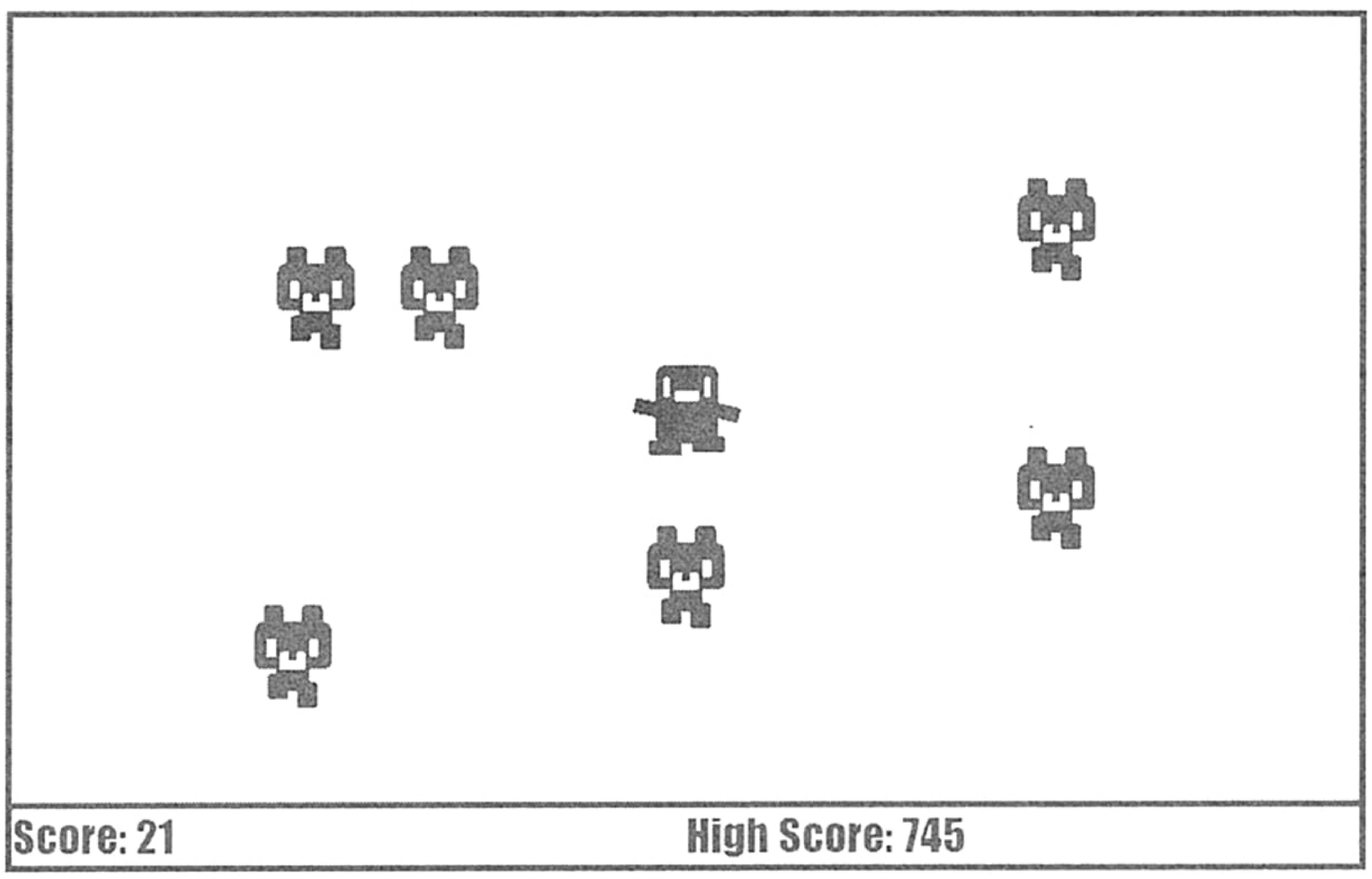
速度をあまり頻繁に変えると、狭い場所で振動するような動きになってしまいます。そこで、低い確率(ここでは5%)で、再び速度をランダムに決めることにしました(Fig.2)。これで敵は決めた動きをしばらく続けますが、そのうちに動きを変えます。

▼ Fig.2: 低い確率で速度を決め直す



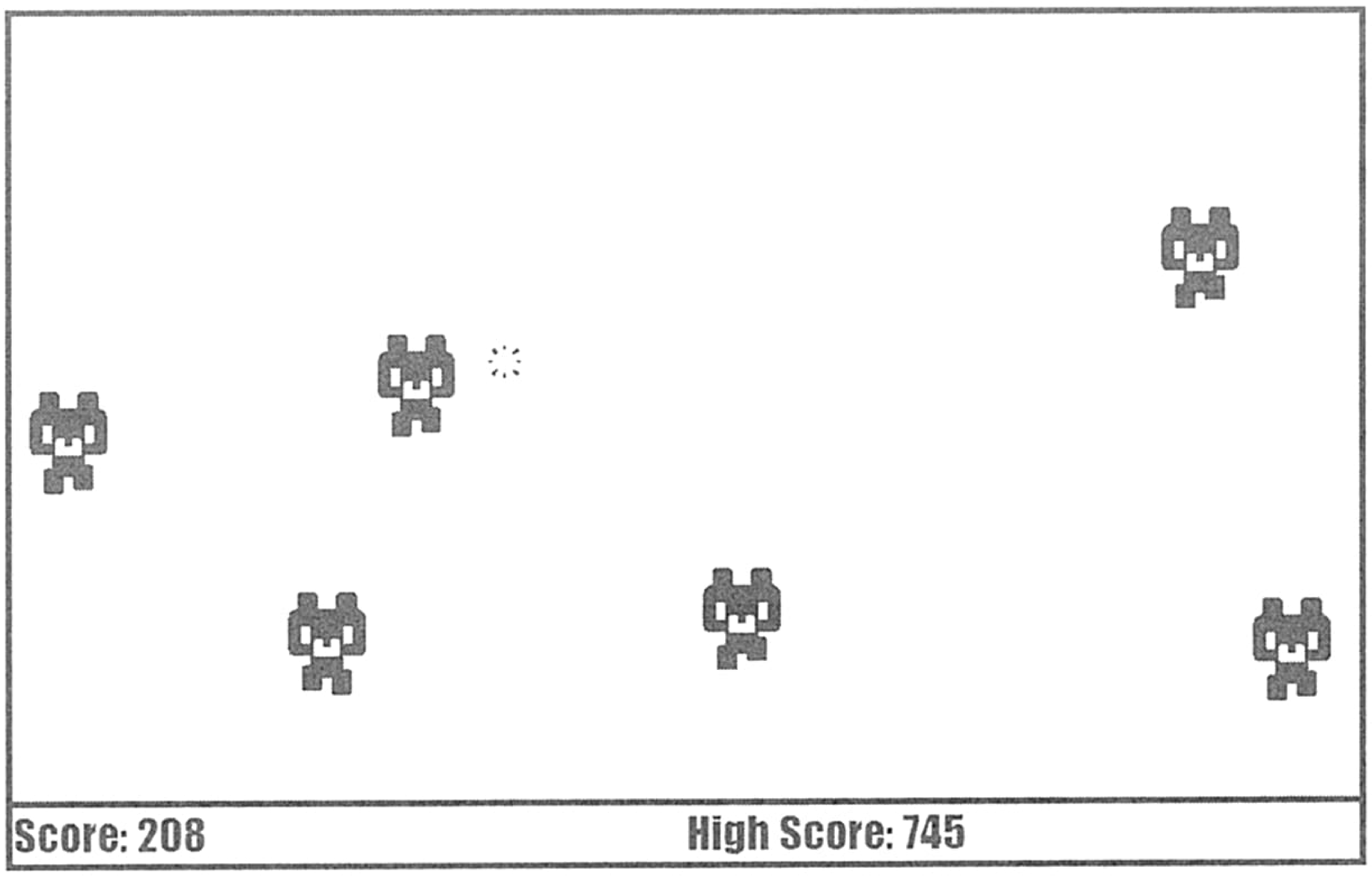
ランダムに動く敵を使って、敵を避けるゲームを作りました(Fig.3)。実行するには、ブラウザでenemyRandom.htmlを開きます。

▼ Fig.3: ランダムに動く敵を避ける



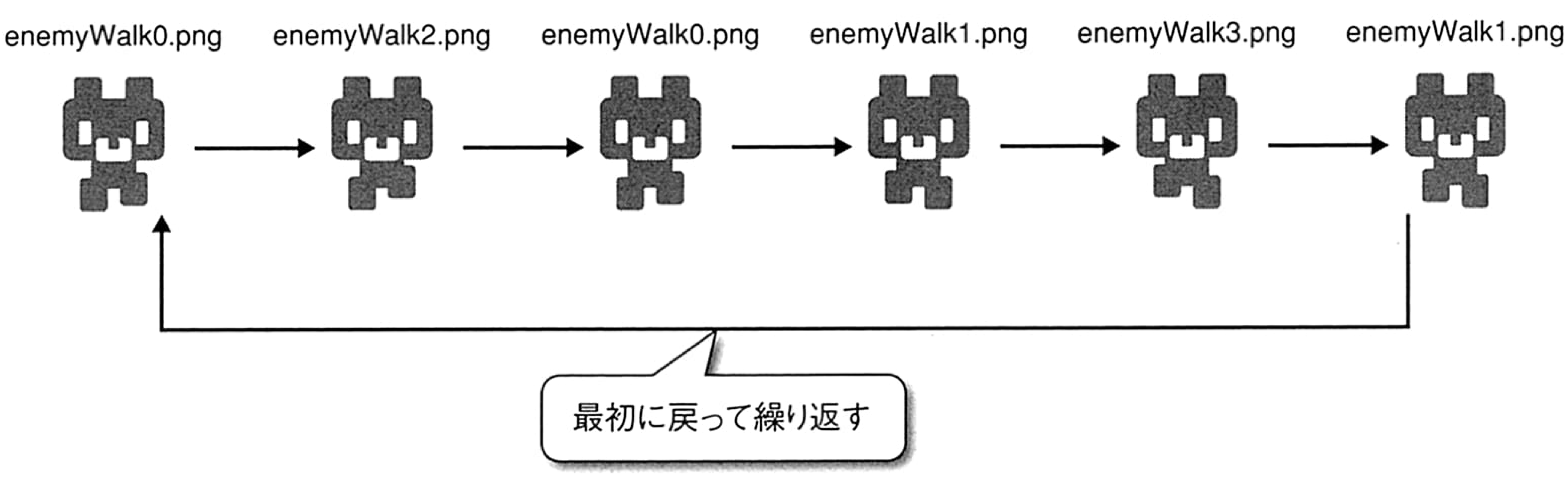
カーソルキーの上下左右で、自機が上下左右に動きます。画面には複数の敵が出現し、ランダムに動きます。自機を上手に動かして、敵に接触しないようにしてください。敵に接触すると自機は消滅します(Fig.4)。生き延びる時間が長いほど、高いスコアを得ることができます。

▼ Fig.4: 敵に接触すると自機は消滅する



なお、敵のアニメーションは自機が歩くアニメーションと同じ要領で作りました(Fig.5)。4枚の画像を使って、6コマのアニメーションを構成しています。

▼ Fig.5: 敵のアニメーション



ランダムに動く敵のプログラム

List1はランダムに動く敵のプログラムです。敵の処理はEnemyオブジェクトにまとめました。最初は、

```
obj.vx=randSign()*ENEMY_SPEED;
obj.vy=randSign()*ENEMY_SPEED;
```

のように敵の速度をランダムに決めます。randSign関数は－1または1をランダムに生成します(p.77のList5)。そのため、速度の方向は左上·左下·右上·右下のいずれかになります。

一定時間ごとに実行する更新処理(onUpdate関数)では、

```
if (randFloat(1)<ENEMY_TURN) {
    vx=randInt2(-1, 2)*ENEMY_SPEED;
    vy=randInt2(-1, 2)*ENEMY_SPEED;
}
```

のようなif文を使って、低い確率(5%)で敵の速度を決め直します。randInt2(－1, 2)は－1, 0, 1のいずれかを生成するので、速度の方向は上下左右斜めに静止を加えた9通りになります。

List1: ランダムに動く敵(enemyRandom.js)

```
// 敵のスピード
var ENEMY_SPEED=0.1;

// 敵が進行方向を変える確率
var ENEMY_TURN=0.05;

// 敵のオブジェクト
function Enemy(x, y) {
  var obj=this;

  // スプライト·アニメーション·当たり判定·座標の初期化
  obj.sprite=game.createSprite();
  obj.sprite.initAnim(enemyWalkAnim);
  obj.sprite.initHit(enemyHit);
  obj.sprite.setXY(x, y);

  // 速度をランダムに決める
  obj.vx=randSign()*ENEMY_SPEED;
  obj.vy=randSign()*ENEMY_SPEED;
```

```
  // 更新処理
  obj.sprite.onUpdate=function() {

    // 座標と速度の取得
    var x=obj.sprite.getX(), y=obj.sprite.getY();
    var vx=obj.vx, vy=obj.vy;

    // 一定の確率で進行方向を変える
    if (randFloat(1)<ENEMY_TURN) {

      // X方向は左右または静止をランダムに選ぶ
      vx=randInt2(-1, 2)*ENEMY_SPEED;

      // Y方向は上下または静止をランダムに選ぶ
      vy=randInt2(-1, 2)*ENEMY_SPEED;
    }

    // 座標に速度を加算する
    x+=vx;
    y+=vy;

    // 画面端で跳ね返る
    if (x<0 || x>XMAX) {
      x-=vx;
      vx=-vx;
    }
    if (y<0 || y>YMAX) {
      y-=vy;
      vy=-vy;
    }

    // 移動中はアニメーションを再生する
    if (x!=obj.sprite.getX() || y!=obj.sprite.getY()) {
      obj.sprite.stepAnim();
    }

    // 座標と速度の更新
    obj.sprite.setXY(x, y);
    obj.vx=vx;
    obj.vy=vy;
  }
}
```

自機を上下左右に動かす

List2は自機のプログラムです。通常状態の自機を動かすalive関数では、

```
if (game.isKey(KEY_LEFT)) x-=PLAYER_SPEED;
if (game.isKey(KEY_RIGHT)) x+=PLAYER_SPEED;
if (game.isKey(KEY_UP)) y-=PLAYER_SPEED;
if (game.isKey(KEY_DOWN)) y+=PLAYER_SPEED;
```

のように、カーソルキーの入力に応じて、自機の座標を変化させます。
自機の座標が変化したら、

```
if (x!=obj.sprite.getX() || y!=obj.sprite.getY()) {
    obj.sprite.stepAnim();
}
```

のように、歩くアニメーションを再生します。自機が立ち止まっているときには、座標が変化しないので、アニメーションの再生が止まります。
敵と接触したかどうかは、

```
if (game.testHitSprite(obj.sprite)) {
    obj.sprite.initAnim(vanishAnim);
    obj.state=STATE_DEAD;
}
```

のように調べます。これは障害物に接触したときと全く同じ処理です(p.103)。接触したら消滅のアニメーションに切り替えて、消滅状態に移行します。

▼ **List2**: 自機を上下左右に動かす(enemyRandom.js)

```
// 自機のスピード
var PLAYER_SPEED=0.2;

// 自機の状態(通常・消滅)
var STATE_ALIVE=0;
var STATE_DEAD=1;

// 自機のオブジェクト
function Player() {
```

```
var obj=this;

// スプライト・アニメーション・当たり判定・座標の初期化
obj.sprite=game.createSprite();
obj.sprite.initAnim(playerWalkAnim);
obj.sprite.initHit(playerHit);
obj.sprite.setXY(XMAX/2, YMAX/2);

// 状態の設定
obj.state=STATE_ALIVE;

// 更新処理
obj.sprite.onUpdate=function() {

  // 状態に応じて分岐する
  switch (obj.state) {
    case STATE_ALIVE: obj.alive(); break;
    case STATE_DEAD: obj.dead(); break;
  }
}

// 通常状態
obj.alive=function() {

  // 座標の取得
  var x=obj.sprite.getX(), y=obj.sprite.getY();

  // カーソルキーの入力に応じて上下左右に移動する
  if (game.isKey(KEY_LEFT)) x-=PLAYER_SPEED;
  if (game.isKey(KEY_RIGHT)) x+=PLAYER_SPEED;
  if (game.isKey(KEY_UP)) y-=PLAYER_SPEED;
  if (game.isKey(KEY_DOWN)) y+=PLAYER_SPEED;

  // 画面端からはみ出ないようにする
  if (x<0) x=0; else if (x>XMAX) x=XMAX;
  if (y<0) y=0; else if (y>YMAX) y=YMAX;

  // 移動中はアニメーションを再生する
  if (x!=obj.sprite.getX() || y!=obj.sprite.getY()) {
    obj.sprite.stepAnim();
  }

  // 座標の更新
  obj.sprite.setXY(x, y);

  // 敵に接触したら、
  // 消滅のアニメーションを再生し、消滅状態にする
```

```
    if (game.testHitSprite(obj.sprite)) {
      obj.sprite.initAnim(vanishAnim);
      obj.state=STATE_DEAD;
    }

    // スコアを加算する
    game.addScore(1);
  }

  // 消滅状態
  obj.dead=function() {

    // アニメーションを再生する
    obj.sprite.stepAnim();

    // アニメーションが終わったらゲームを初期化する
    if (obj.sprite.isAnimOver()) {
      init();
    }
  }
}
```

ランダムに敵が動くゲームの本体

List3はランダムに動く敵を避けるゲームの本体です。ゲームの初期化を行うinit関数と、メインルーチンのmain関数を定義します。

init関数では画面を消去し、自機と敵のオブジェクトを作成します。main関数では、ゲーム本体のオブジェクト、アニメーション、当たり判定を作成します。自機と敵の当たり判定は、ゲームを遊びやすくするために、見た目よりも少し小さくしてあります。

List3: ランダムに動く敵を避けるゲームの本体(enemyRandom.js)

```
// ゲーム·自機·敵のオブジェクト
var game;
var player;
var enemy;

// 自機·敵·消滅のアニメーション
var playerWalkAnim;
var enemyWalkAnim;
var vanishAnim;
```

```
// 自機·敵の当たり判定
var playerHit;
var enemyHit;

// ゲームの初期化
function init() {

  // 画面の消去
  game.clear();

  // スコアの初期化
  game.setScore(0);

  // 自機の作成
  player=new Player();

  // 敵の作成
  enemy=new Array();
  enemy[0]=new Enemy(0, 0);
  enemy[1]=new Enemy(XMAX/2, 0);
  enemy[2]=new Enemy(XMAX, 0);
  enemy[3]=new Enemy(0, YMAX)
  enemy[4]=new Enemy(XMAX/2, YMAX)
  enemy[5]=new Enemy(XMAX, YMAX);
}

// メインルーチン
function main() {

  // ゲーム本体の作成
  game=new Game();

  // アニメーションの作成
  playerWalkAnim=new Array(
    "pcWalk0.png", "pcWalk2.png", "pcWalk0.png",
    "pcWalk1.png", "pcWalk3.png", "pcWalk1.png"
  );
  enemyWalkAnim=new Array(
    "enemyWalk0.png", "enemyWalk2.png", "enemyWalk0.png",
    "enemyWalk1.png", "enemyWalk3.png", "enemyWalk1.png"
  );
  vanishAnim=new Array(
    "void.png", "vanish8.png", "vanish6.png",
    "vanish4.png", "vanish2.png", "vanish0.png",
    "vanish1.png", "vanish2.png", "vanish3.png",
    "vanish4.png", "vanish5.png", "vanish6.png",
```

```
    "vanish7.png", "vanish8.png", "void.png"
  );

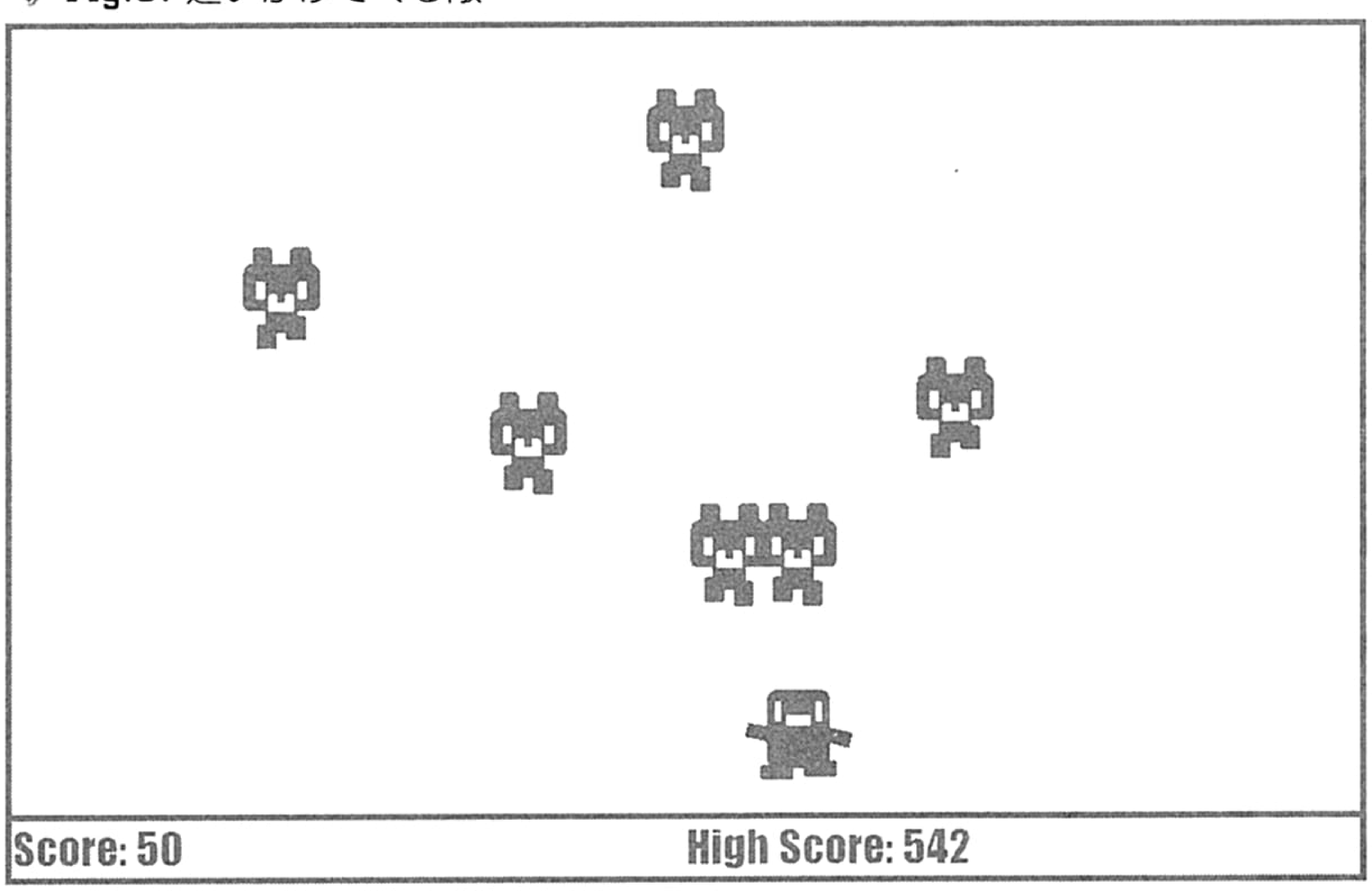
  // 当たり判定の作成
  playerHit=new Array(0.25, 0.25, 0.75, 0.75);
  enemyHit=new Array(0.25, 0.25, 0.75, 0.75);

  // ゲームの初期化
  init();
}
```

追いかけてくる敵

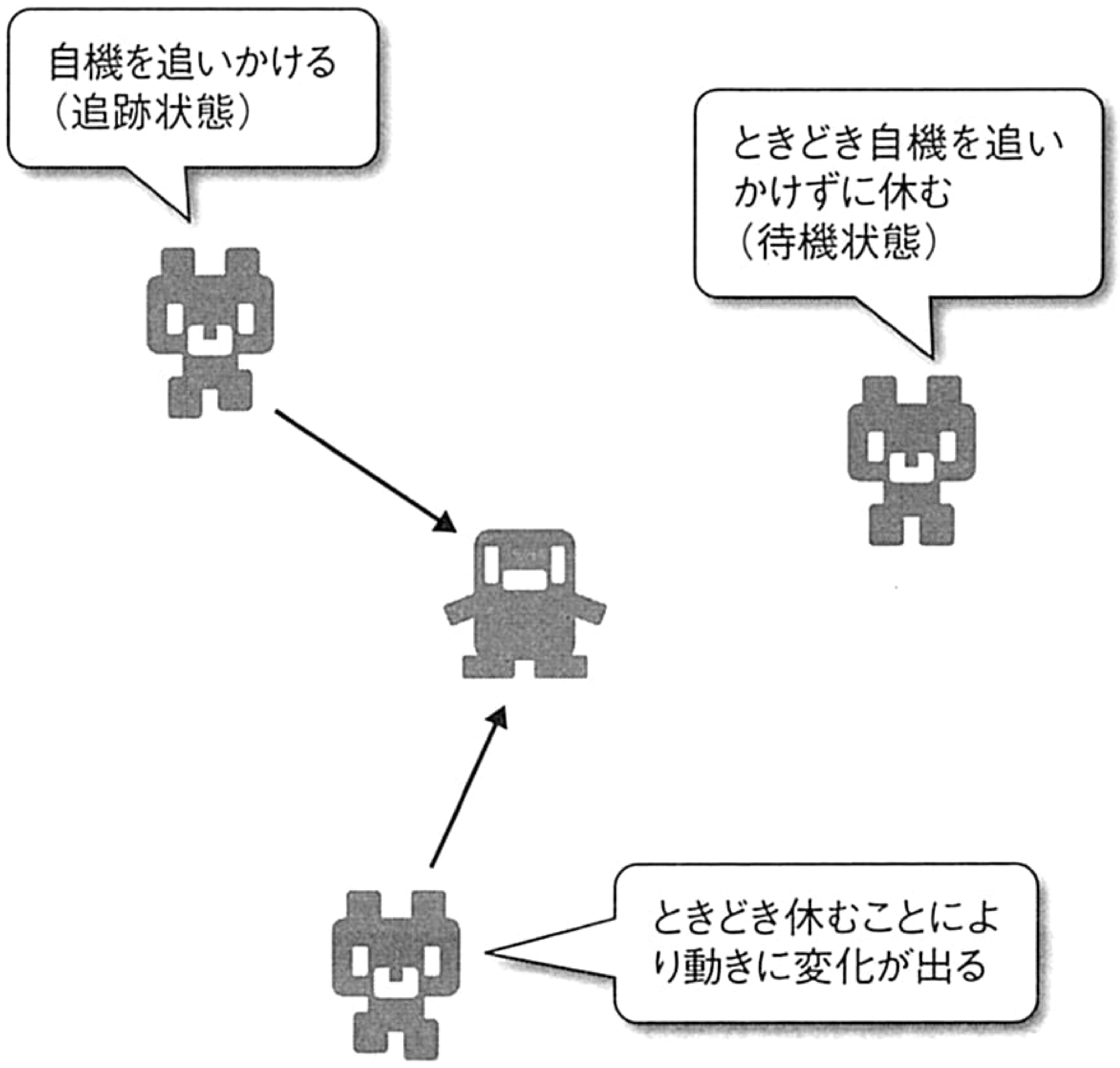
ランダムに動く敵の次は、追いかけてくる敵を作りましょう。追いかけてくる敵のサンプルを実行するには、ブラウザでenemyChase.htmlを開きます(Fig.6)。自機をカーソルキーで上下左右に動かして、敵に接触しないように逃げ回ってください。

▼ **Fig.6**: 追いかけてくる敵



追いかけてくる敵を面白くするには、自機をずっと追いかけさせ続けるのではなく、ときどき休ませます(Fig.7)。自機を追いかけてばかりでは、動きが単調になります。また複数の敵がいるときには、全ての敵が同じ動きになってしまい、複数の敵がいる意味がなくなってしまいます。

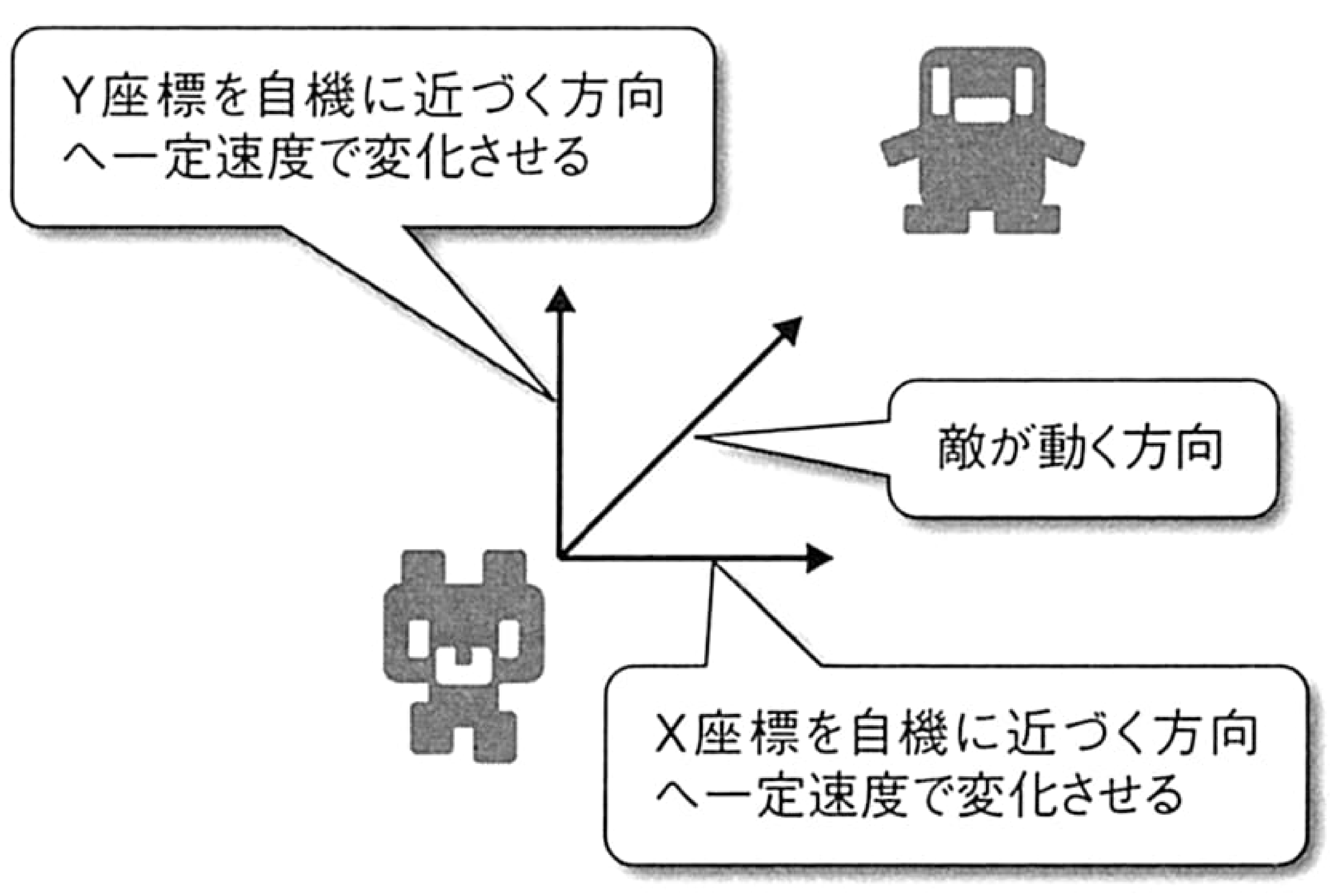
▼ Fig.7: 追いかけてくる敵の動き



そこで、自機を追いかける追跡状態と、その場で立ち止まる待機状態という2つの状態を用意します。追跡状態と待機状態の間を低い確率で移行させれば、しばらく追いかけると立ち止まり、また少しすると動き出すといった動きが作れます。追跡から待機に移行する確率と、待機から追跡に移行する確率を変えれば、よく追いかける敵にも、休んでばかりいる敵にもなります。

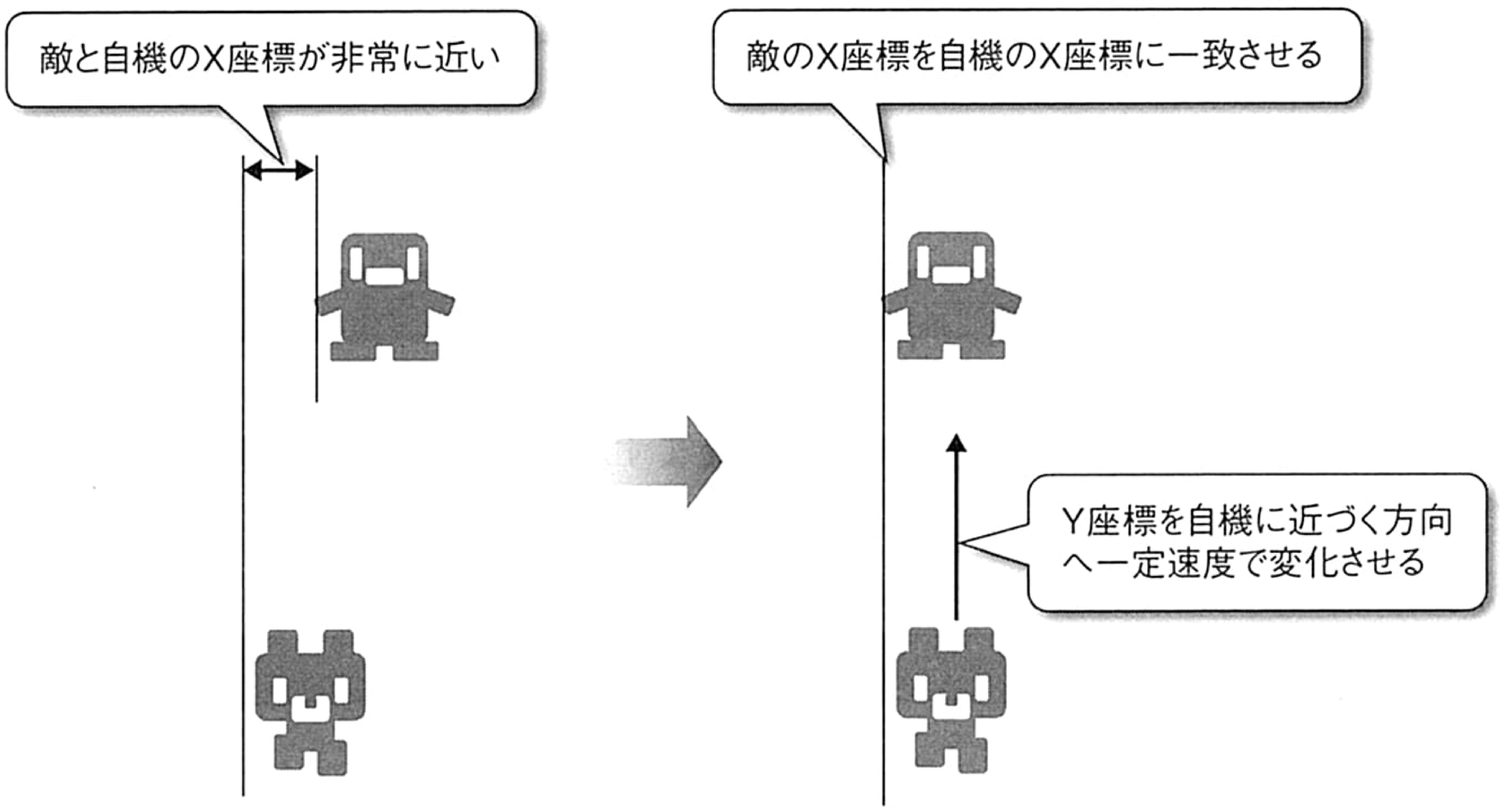
追跡状態では、敵と自機の座標を比べて、自機に近づくように敵を動かします(Fig.8)。敵のX座標とY座標を、それぞれ自機に近づく方向へ一定速度で変化させます。

▼ Fig.8: 敵を自機に近づける



敵と自機のX座標またはY座標が非常に近いときには、一定速度で移動させるのではなく、敵のX座標またはY座標を自機の座標に一致させます(Fig.9)。これはマウスカーソルを追いかける自機(p.121)と同様に、敵と自機が近いときに敵を一定速度で動かすと、振動するような動きをしてしまうためです。

Fig.9: 敵と自機が非常に近いとき



追いかけてくる敵のプログラム

List4はランダムに動く敵のプログラムです。Enemyオブジェクトを定義しました。

まず、追跡状態か待機状態かを表すために、

```
obj.chase=true;
```

というプロパティを定義しました。trueならば追跡状態、falseならば待機状態です。

更新処理では、

```
var x=obj.sprite.getX(), y=obj.sprite.getY();
var px=player.sprite.getX(), py=player.sprite.getY();
```

のように、敵と自機の座標を取得します。そして追跡状態ならば、

```
if (Math.abs(px-x)<ENEMY_SPEED) {
```

のように、敵と自機のX座標が近いかどうかを調べます。近いときには、

```
x=px;
```

として、敵を自機のX座標に移動させます。遠いときには、

```
if (px<x) x-=ENEMY_SPEED; else x+=ENEMY_SPEED;
```

として、自機に近づく方向へ一定速度で移動させます。Y座標についても同様です。
　また、

```
if (randFloat(1)<ENEMY_WAIT) obj.chase=false;
```

のように、低い確率(ここでは2%)で待機状態に移行します。逆に待機状態では、

```
if (randFloat(1)<ENEMY_CHASE) obj.chase=true;
```

のように、やはり低い確率(ここでは4%)で追跡状態に移行します。確率を微妙に変えてあるので、敵は待機よりも追跡を優先します。

List4: 追いかけてくる敵(enemyChase.js)

```
// 敵のスピード
var ENEMY_SPEED=0.1;

// 敵が追跡状態になる確率
var ENEMY_CHASE=0.04;

// 敵が待機状態になる確率
var ENEMY_WAIT=0.02;

// 敵のオブジェクト
function Enemy(x, y) {
  var obj=this;

  // スプライト・アニメーション・当たり判定・座標の初期化
```

```
obj.sprite=game.createSprite();
obj.sprite.initAnim(enemyWalkAnim);
obj.sprite.initHit(enemyHit);
obj.sprite.setXY(x, y);

// 追跡状態(true)か待機状態(false)か
obj.chase=true;

// 更新処理
obj.sprite.onUpdate=function() {

  // 敵と自機の座標を取得する
  var x=obj.sprite.getX(), y=obj.sprite.getY();
  var px=player.sprite.getX(), py=player.sprite.getY();

  // 追跡状態
  if (obj.chase) {

    // 敵と自機のX座標が近いときには、
    // 自機と同じX座標に移動する
    if (Math.abs(px-x)<ENEMY_SPEED) {
      x=px;
    } else

    // 敵と自機のX座標が遠いときには、
    // 自機に近づく方向へ一定速度で移動する
    {
      if (px<x) x-=ENEMY_SPEED; else x+=ENEMY_SPEED;
    }

    // 敵と自機のY座標が近いときには、
    // 自機と同じY座標に移動する
    if (Math.abs(py-y)<ENEMY_SPEED) {
      y=py;
    } else

    // 敵と自機のY座標が遠いときには、
    // 自機に近づく方向へ一定速度で移動する
    {
      if (py<y) y-=ENEMY_SPEED; else y+=ENEMY_SPEED;
    }

    // 一定確率で待機状態に移行する
    if (randFloat(1)<ENEMY_WAIT) obj.chase=false;
  } else

  // 待機状態
```

```
    {
      // 移動はせずに、一定確率で追跡状態に移行する
      if (randFloat(1)<ENEMY_CHASE) obj.chase=true;
    }

    // 移動中はアニメーションを再生する
    if (x!=obj.sprite.getX() || y!=obj.sprite.getY()) {
      obj.sprite.stepAnim();
    }

    // 座標の更新
    obj.sprite.setXY(x, y);
  }
}
```

Chapter 6

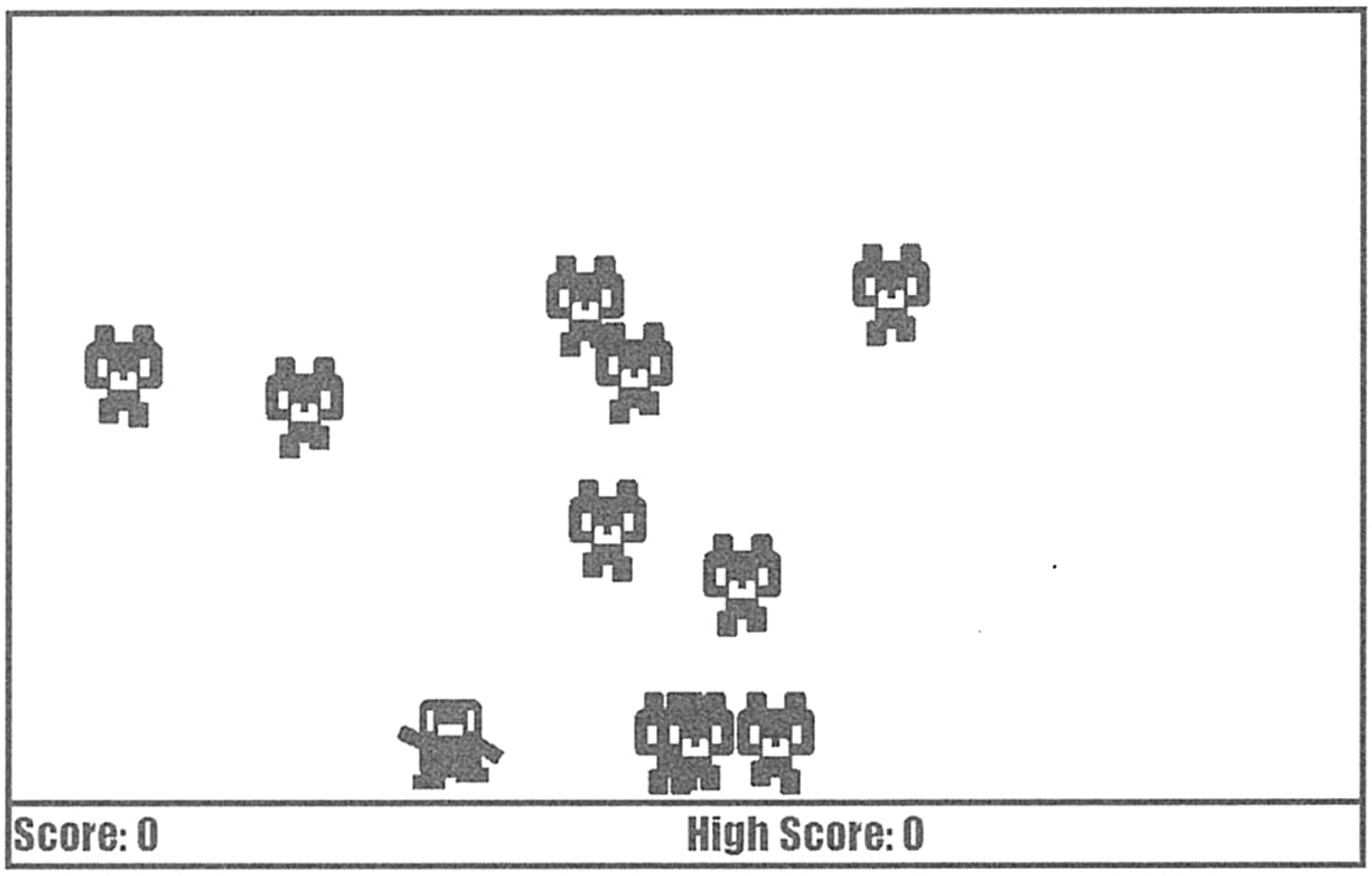
武器

武器で敵を撃つ

第5章では敵を避けるゲームを作りましたが、敵から逃げ回るだけでは物足りません。今度は武器を発射して敵に反撃できる、シューティングゲームを作りましょう(Fig.1)。このゲームを実行するには、shootEnemy.htmlをブラウザで開きます。

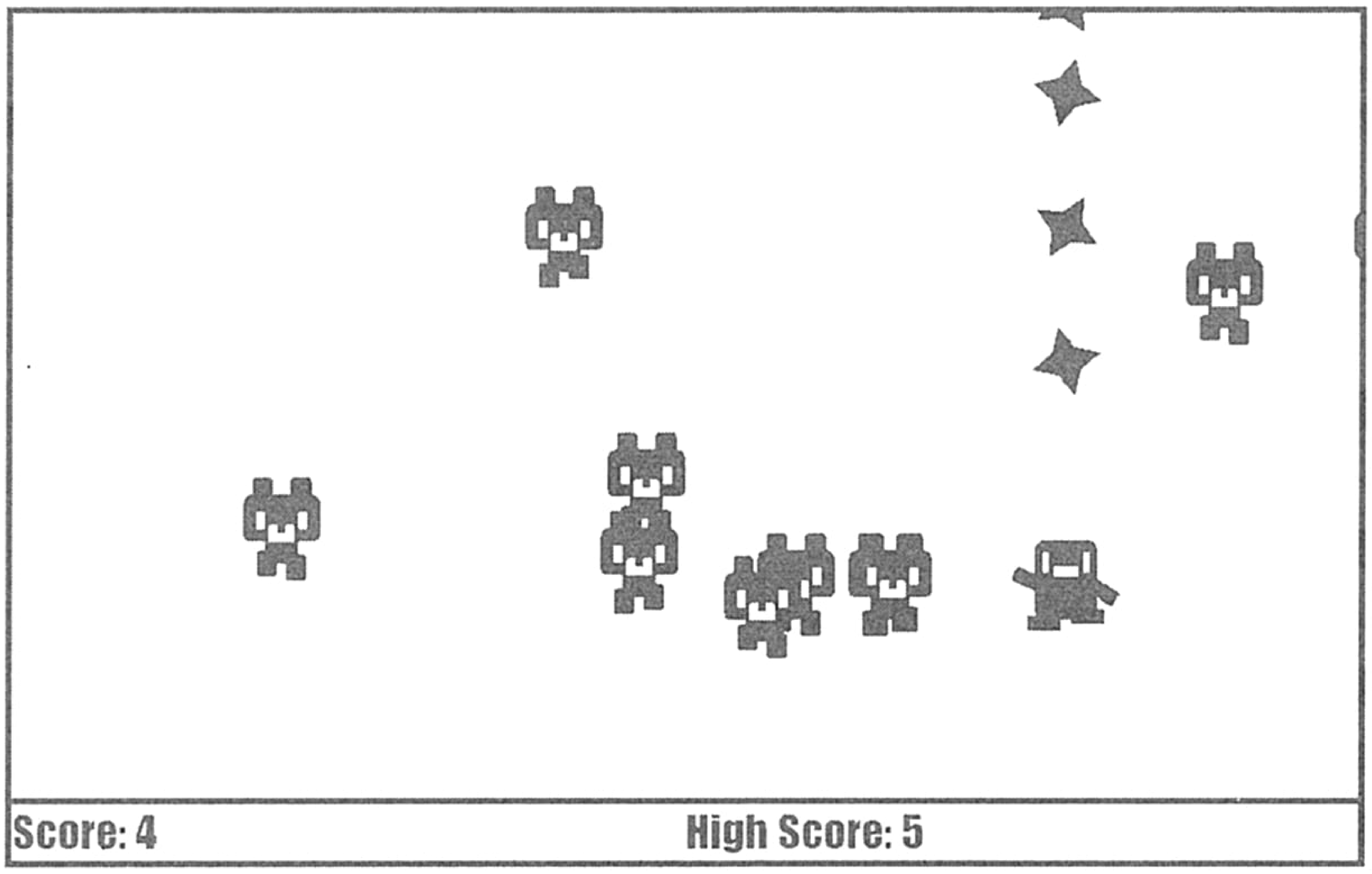
自機はカーソルキーで上下左右に動きます。周囲から敵が押し寄せてくるので、敵に当たらないように避けてください。敵に接触すると、自機は消滅してしまいます。

▼ Fig.1: シューティングゲーム



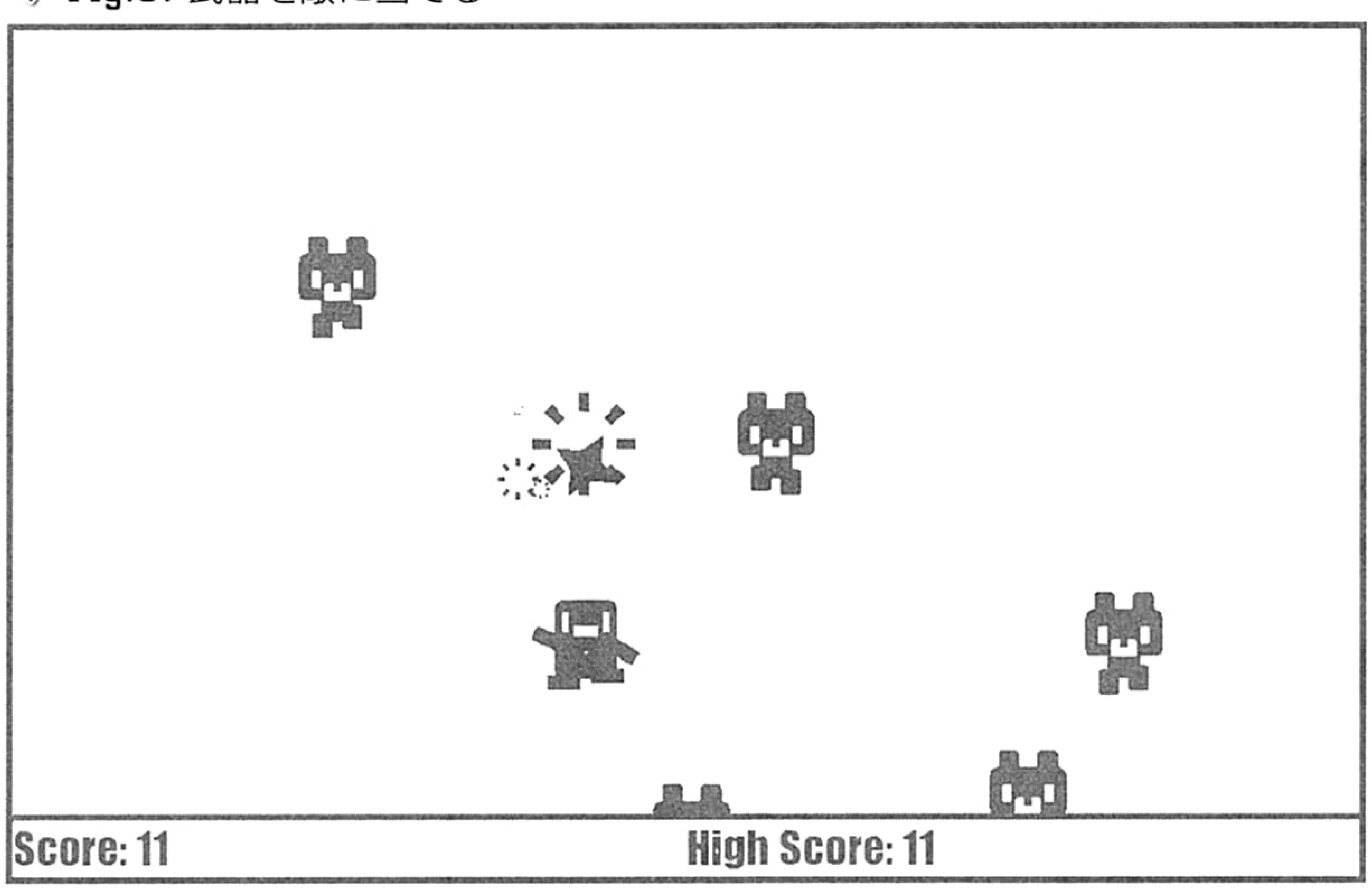
スペースキーで武器(手裏剣)を撃つことができます(Fig.2)。武器は画面上方へ飛んでいきます。一度に画面上に出現する武器は最大5個です。プログラムとしては武器をいくつでも出せるのですが、少し火力が弱い方が面白いので、あえて出現数を少なめにしました。

▼ **Fig.2**: 武器を撃つ



武器を敵に当てると、敵を消滅させることができます(Fig.3)。敵を1体倒すごとに、スコアが1点入ります。押し寄せる敵の勢いに負けずに、素早く武器を発射して、できるだけ数多くの敵を倒してください。

▼ **Fig.3**: 武器を敵に当てる

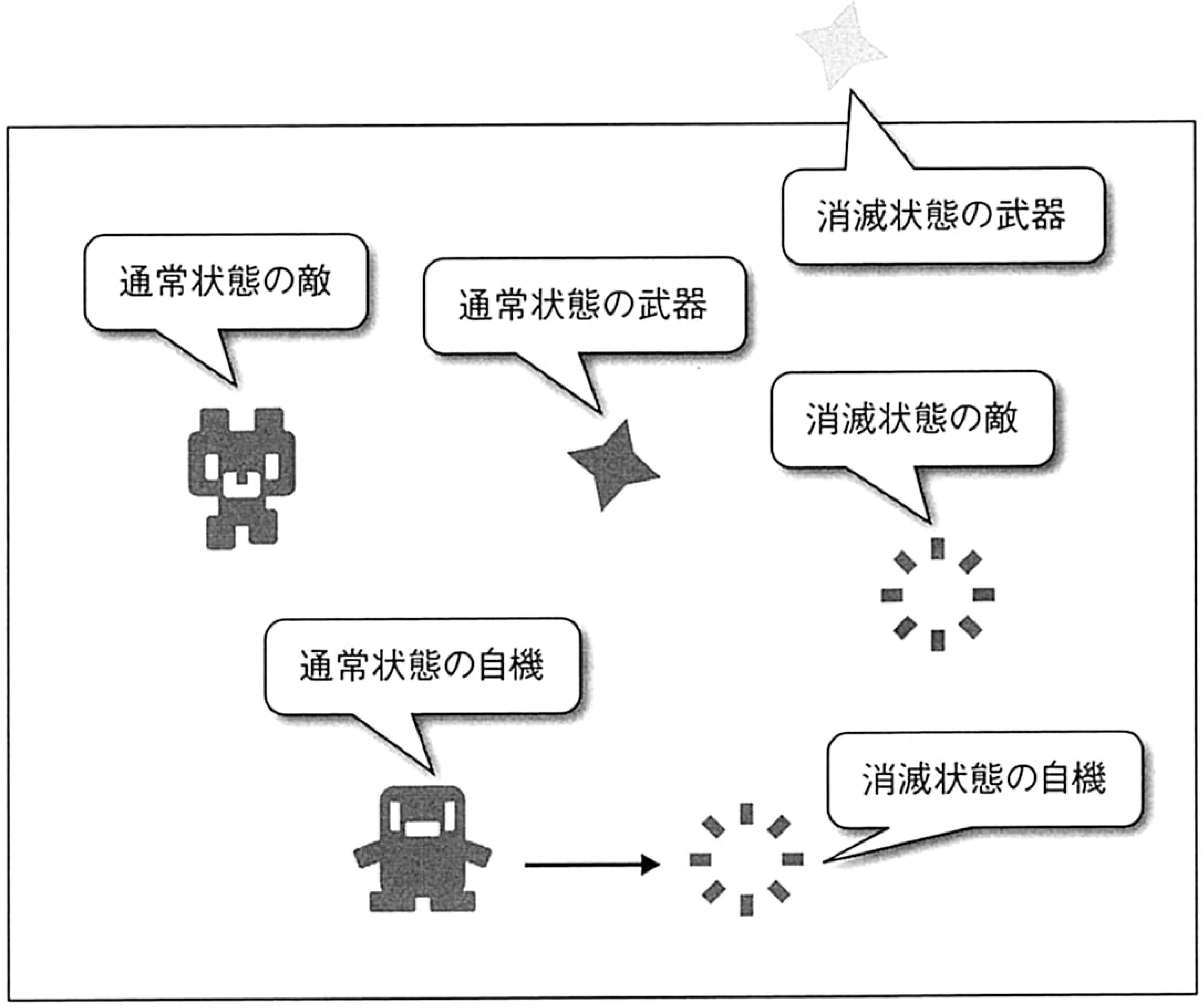


状態の管理

このゲームには、自機·敵·武器という3種類の物体が登場します。そしてこれらの物体には、それぞれ通常状態と消滅状態があります(Fig.4)。

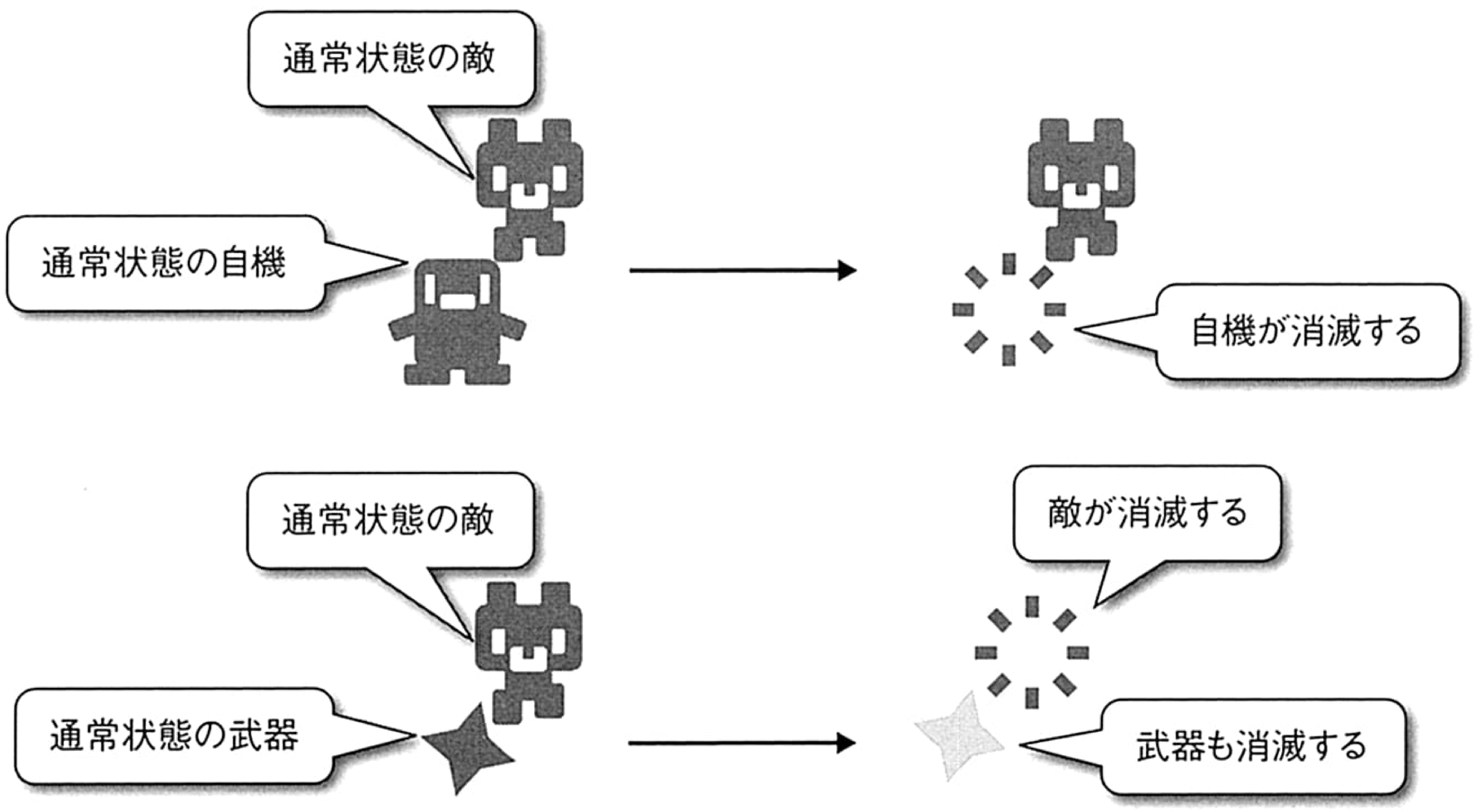
- **通常状態の自機**
  カーソルキーで上下左右に動かせる自機です。敵に接触すると消滅状態になります。
- **消滅状態の自機**
  消滅しつつある自機です。消滅のアニメーションが終わるとゲーム終了です。
- **通常状態の敵**
  自機を追いかけてくる敵です。武器に接触すると消滅状態になります。
- **消滅状態の敵**
  消滅しつつある敵です。消滅のアニメーションが終わると、新しい位置に再び出現します。
- **通常状態の武器**
  発射されて画面上方に飛んでいく武器です。敵に接触するか、画面外に出ると消滅状態になります。
- **消滅状態の武器**
  消滅した武器です。初期化することによって、再び発射することができます。

▼ **Fig.4**: 物体の状態



これらの状態は、当たり判定処理にも深く関わってきます(Fig.5)。例えば、自機は通常状態の敵に当たったときだけ消滅し、敵は通常状態の武器に当たったときだけ消滅します。当たり判定処理の際には、どの状態の物体同士が接触したのかを調べて、組み合わせに応じて処理を変える必要があります。

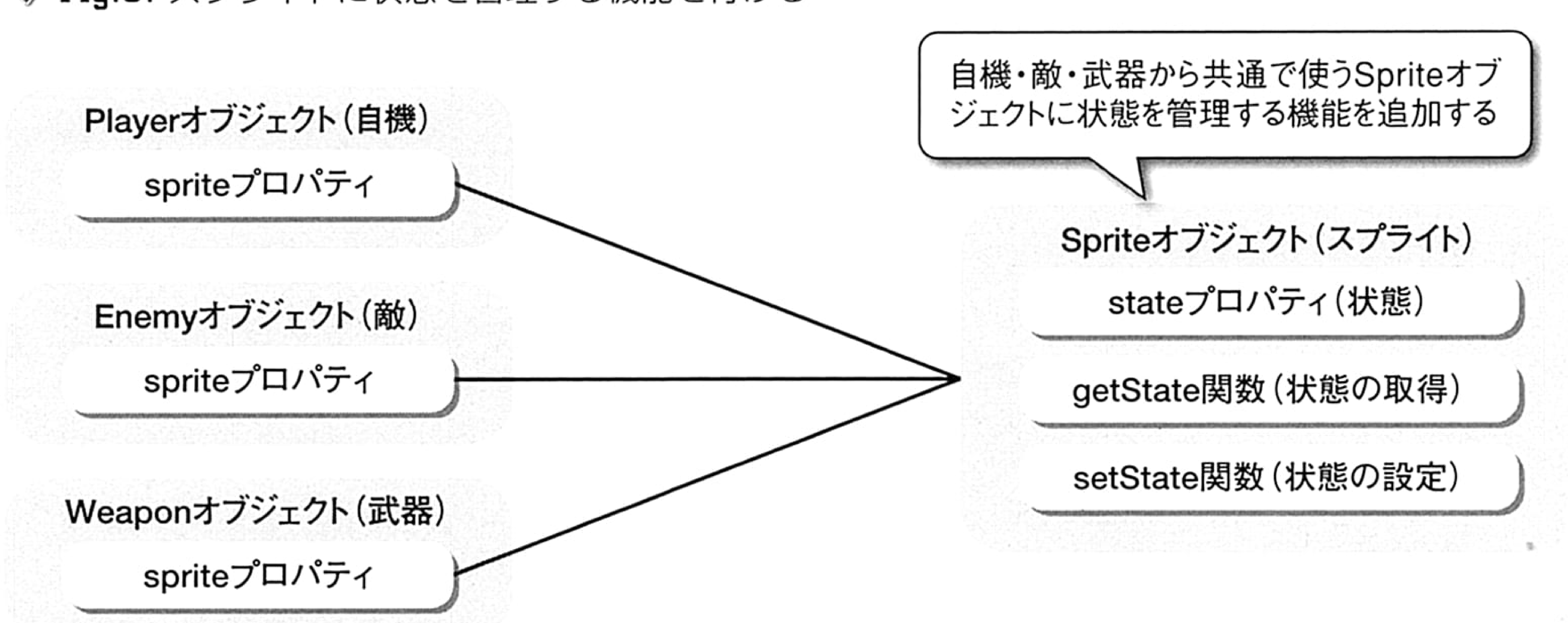
▼ Fig.5: 物体の状態と当たり判定処理



そこで、自機・敵・武器といった全ての物体について、状態を管理することにします。前章までは、自機についてはstateプロパティで状態を管理していました。同様に、敵や武器についてもstateプロパティを追加すればよいでしょう。

しかし、同じプロパティを3種類のオブジェクトに追加するのは煩雑です。そこで、自機・敵・武器で共通に使うスプライト(Spriteオブジェクト)に、状態を管理する機能を追加することにしました(Fig.6)。

▼ Fig.6: スプライトに状態を管理する機能を付ける



状態を管理するプログラム

List1はスプライトオブジェクトに追加した状態管理の機能です。まず、

```
obj.state=0;
```

のように、状態を管理するstateプロパティを定義しました。さらに、状態を操作するために、

- getState関数：状態の取得
- setState関数：状態の設定

という2つの関数を定義しました。

▼ **List1**: 状態の管理(common.js)

```
// スプライト
function Sprite(div, zIndex) {
  // ...(中略)...

  // 状態を保持するプロパティ
  obj.state=0;

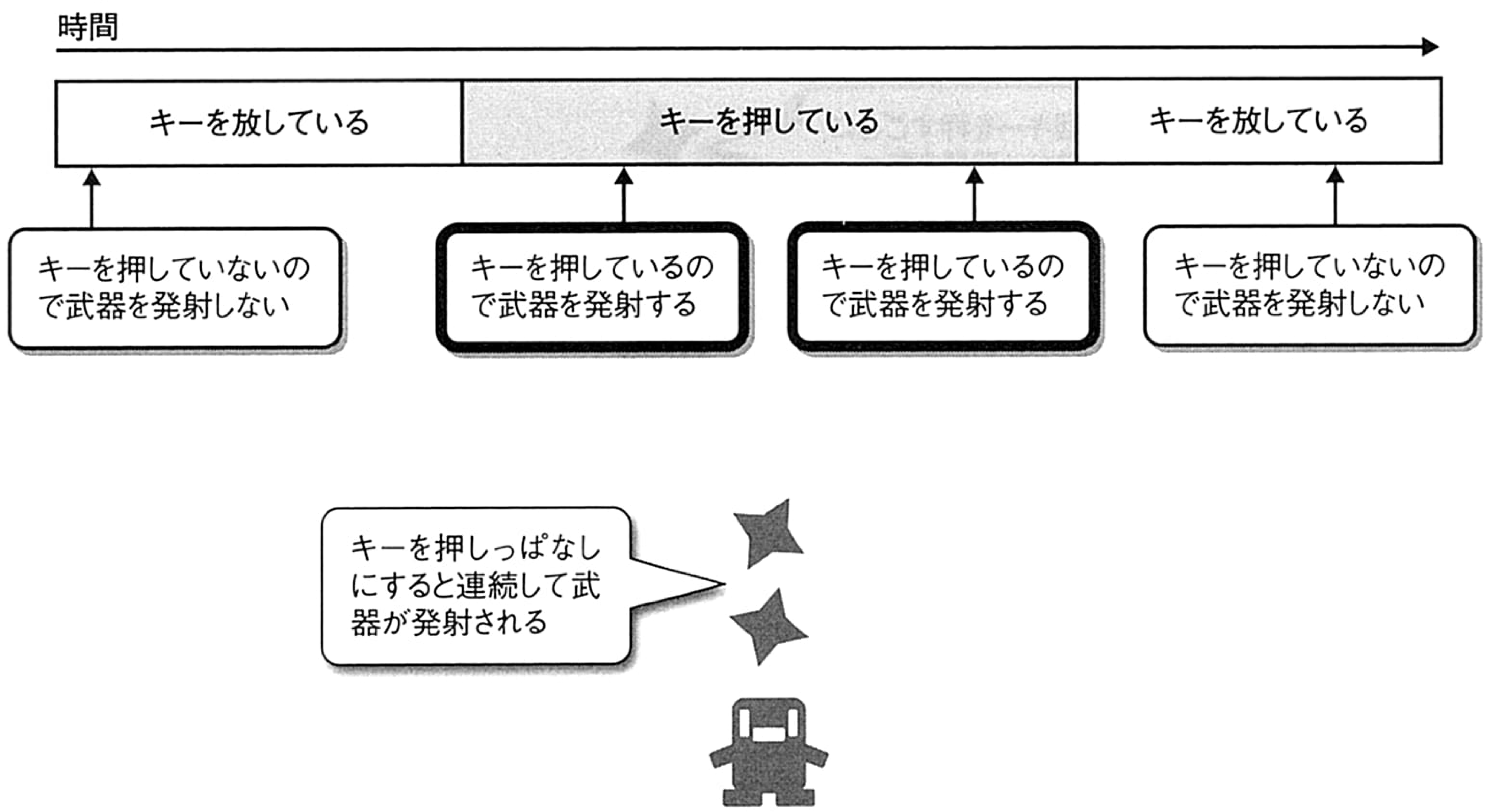
  // 状態の取得
  obj.getState=function() {
    return obj.state;
  }

  // 状態の設定
  obj.setState=function(state) {
    obj.state=state;
  }
}
```

武器の発射

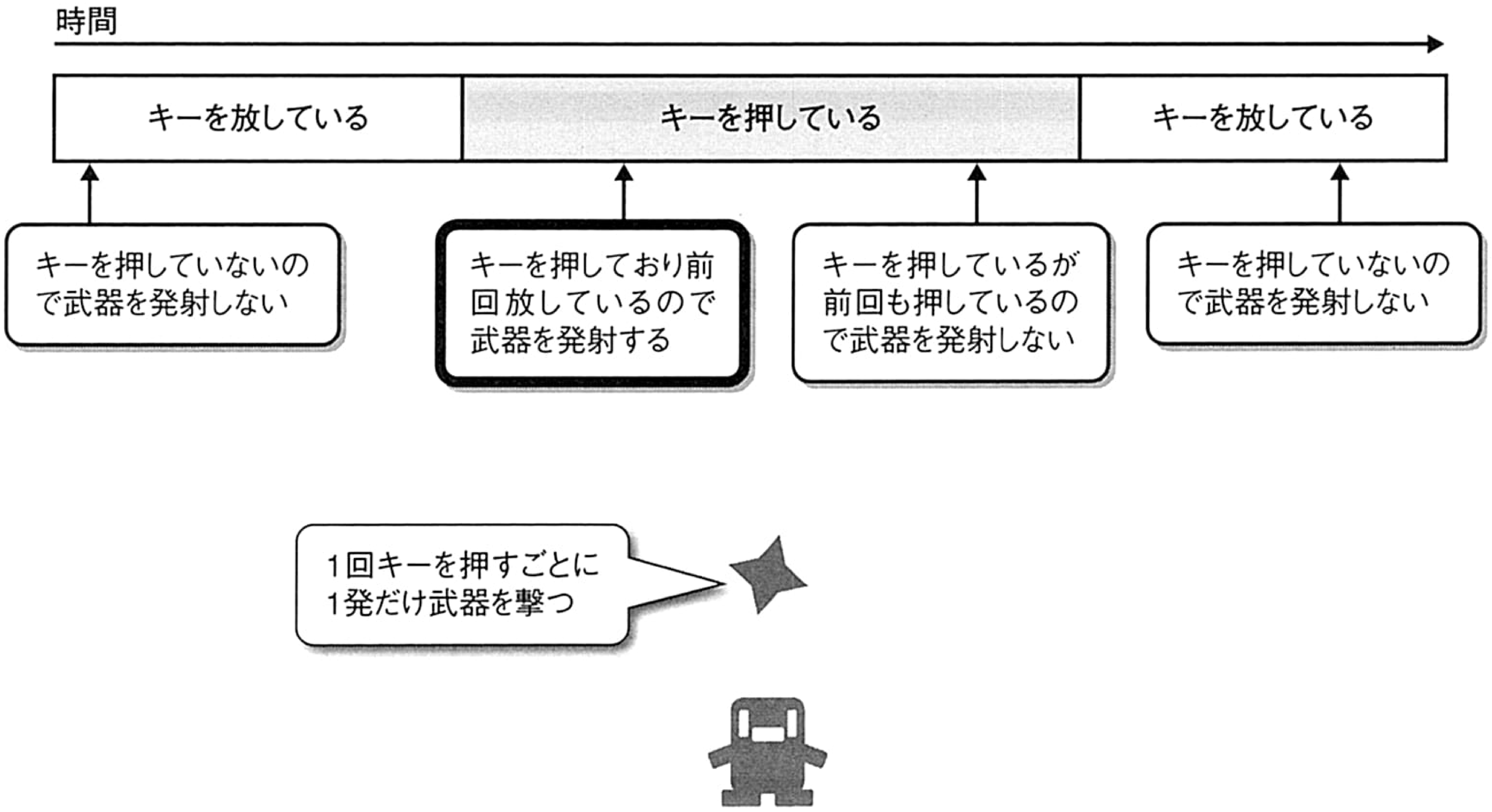
スペースキーを押すと、自機が武器を発射します。これは、スペースキーを押しているかどうかを調べて、押していたら武器を発射すれば実現できます。ただしこれだけでは、スペースキーを押しっぱなしにしたときに、武器が連続して発射されます(Fig.7)。

▼ **Fig.7**: 武器が連続して発射される



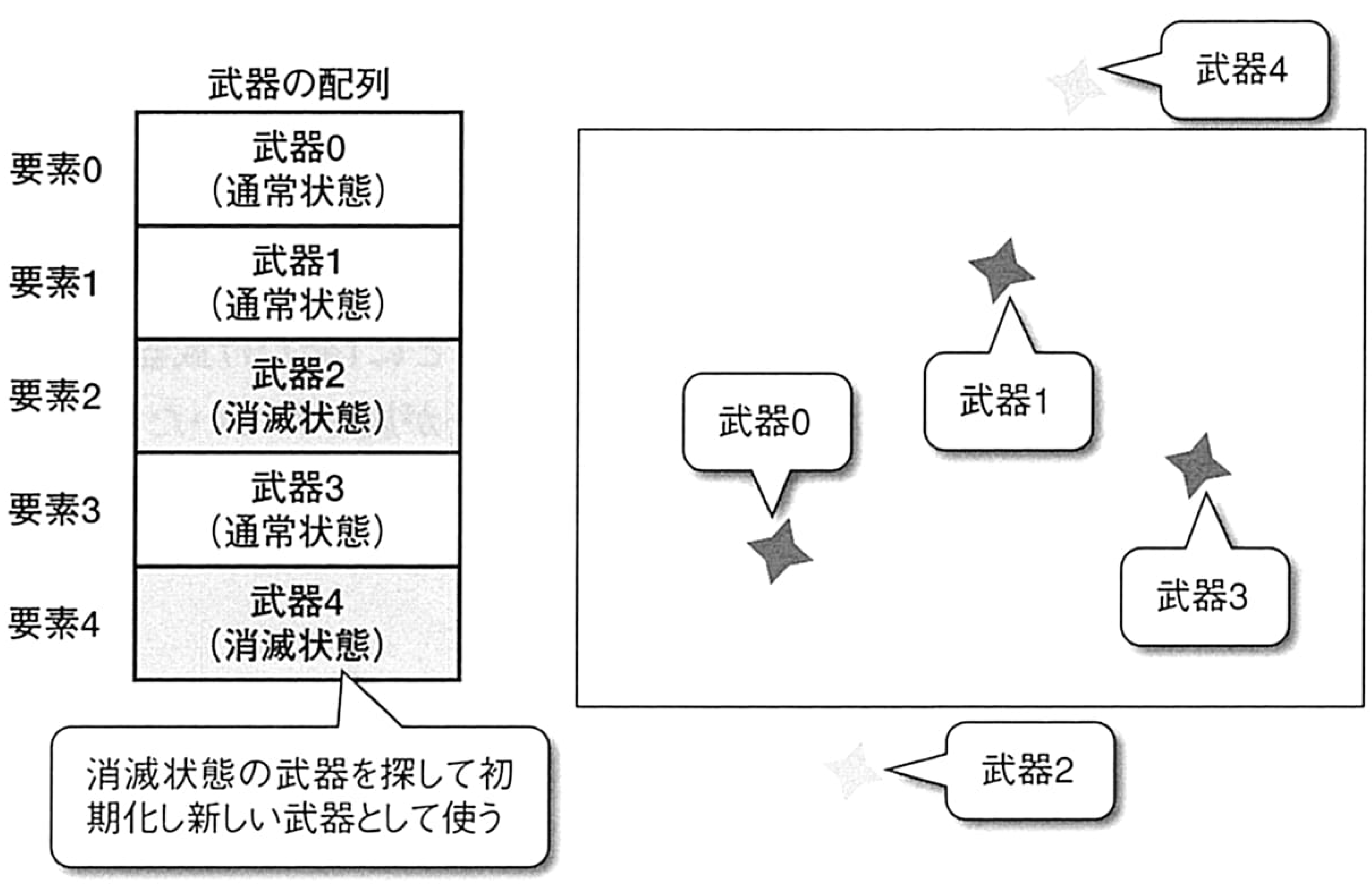
今回のサンプルでは、キーを押しっぱなしで武器が出るよりも、キーを連打して武器を連射した方が面白そうでした。このようにスペースキーを1回押すごとに1発だけ武器を撃つには、スペースキーを押していて、かつ前回調べたときにスペースキーが放されていたときだけ、武器を発射します(Fig.8)。この方式を実現するには、調べたスペースキーの状態を変数に保存しておき、次回調べるときに参照します。

▼ **Fig.8**: 1回キーを押すごとに1発だけ発射する



武器を発射するには、武器のオブジェクトを初期化する必要があります。武器のオブジェクトは配列で管理します(Fig.9)。

▼ **Fig.9**: 武器の配列



武器には通常状態と消滅状態があります。消滅状態の武器とは、敵に当たったり画面外に出たりして、もう使わなくなった武器です。この消滅状態の武器を配列から探して、座標などを初期化することにより、新しい武器として使います。

武器を発射する自機のプログラム

武器を発射する処理は、自機オブジェクトのプログラムに組み込みます(List2)。この自機はカーソルキーで上下左右に動き、スペースキーを押すと武器を発射します。

自機の移動に関する処理は、敵を避ける自機のプログラム(p.141のList2)とほぼ同じです。ここでは武器を撃つ処理について解説します。

まず、スペースキーの状態を保存するために、

```
obj.prevKey=true;
```

のようなprevKeyプロパティを定義します。このプロパティには、スペースキーを押したときはtrue、放したときはfalseを記録します。

武器の発射には、

```
if (!obj.prevKey && game.isKey(KEY_SPACE)) {
```

のようなif文を使います。前回調べたときにスペースキーを放しており、今回スペースキーを押しているときに限って、武器を発射します。

武器を発射するために、

```
for (i=0; i<weapon.length; i++) {
    if (weapon[i].sprite.getState()==STATE_WEAPON_DEAD) {
        weapon[i].init(x, y, 0, -1);
        break;
    }
}
```

のようなfor文を使って、武器の配列の中から、消滅状態の武器を探します。見つかったら、その武器を初期化して、新しい武器として発射します。武器を初期化する処理(init関数)については後述します。

なお、break文はfor文の繰り返しを中断するための構文です。この例では、消滅状態の武器が1個見つかれば十分なので、見つかった時点で武器を探す処理を中断しています。

消滅状態の武器が見つからなければ、武器は発射しません。これは既に画面上に最大数(5個)の武器が出ている状況です。

最後に、

```
obj.prevKey=game.isKey(KEY_SPACE);
```

のように、スペースキーの状態を保存します。これは次の武器を発射するときに、前回スペースキーを放していたかどうかを調べるために使います。

▼ **List2**: 武器を発射する自機(shootEnemy.js)

```
// 自機のスピード
var PLAYER_SPEED=0.2;

// 自機の状態(通常・消滅)
var STATE_PLAYER_ALIVE=0;
var STATE_PLAYER_DEAD=1;

// 自機のオブジェクト
function Player() {
  var obj=this;

  // スプライト・アニメーション・当たり判定・座標の初期化
  obj.sprite=game.createSprite();
  obj.sprite.initAnim(playerWalkAnim);
  obj.sprite.initHit(playerHit);
  obj.sprite.setXY(XMAX/2, YMAX/2);

  // 最初は通常状態にしておく
  obj.sprite.setState(STATE_PLAYER_ALIVE);

  // 前回スペースキーを押したかどうか
  obj.prevKey=true;

  // 更新処理
  obj.sprite.onUpdate=function() {

    // 状態に応じて分岐する
    switch (obj.sprite.getState()) {
      case STATE_PLAYER_ALIVE: obj.alive(); break;
      case STATE_PLAYER_DEAD: obj.dead(); break;
    }
  }

  // 通常状態
  obj.alive=function() {

    // 座標の取得
    var x=obj.sprite.getX(), y=obj.sprite.getY();
```

```
    // カーソルキーの入力に応じて上下左右に移動する
    if (game.isKey(KEY_LEFT)) x-=PLAYER_SPEED;
    if (game.isKey(KEY_RIGHT)) x+=PLAYER_SPEED;
    if (game.isKey(KEY_UP)) y-=PLAYER_SPEED;
    if (game.isKey(KEY_DOWN)) y+=PLAYER_SPEED;

    // 画面端からはみ出ないようにする
    if (x<0) x=0; else if (x>XMAX) x=XMAX;
    if (y<0) y=0; else if (y>YMAX) y=YMAX;

    // 移動中はアニメーションを再生する
    if (x!=obj.sprite.getX() || y!=obj.sprite.getY()) {
      obj.sprite.stepAnim();
    }

    // 座標の更新
    obj.sprite.setXY(x, y);

    // 前回スペースキーを押しておらず、今回押した場合
    if (!obj.prevKey && game.isKey(KEY_SPACE)) {

      // 消滅状態の武器を探す
      for (i=0; i<weapon.length; i++) {
        if (weapon[i].sprite.getState()==STATE_WEAPON_DEAD) {

          // 消滅状態の武器が見つかったら発射する
          weapon[i].init(x, y, 0, -1);
          break;
        }
      }
    }

    // 今回スペースキーを入力したかどうかを保存する
    obj.prevKey=game.isKey(KEY_SPACE);
  }

  // 消滅状態
  obj.dead=function() {

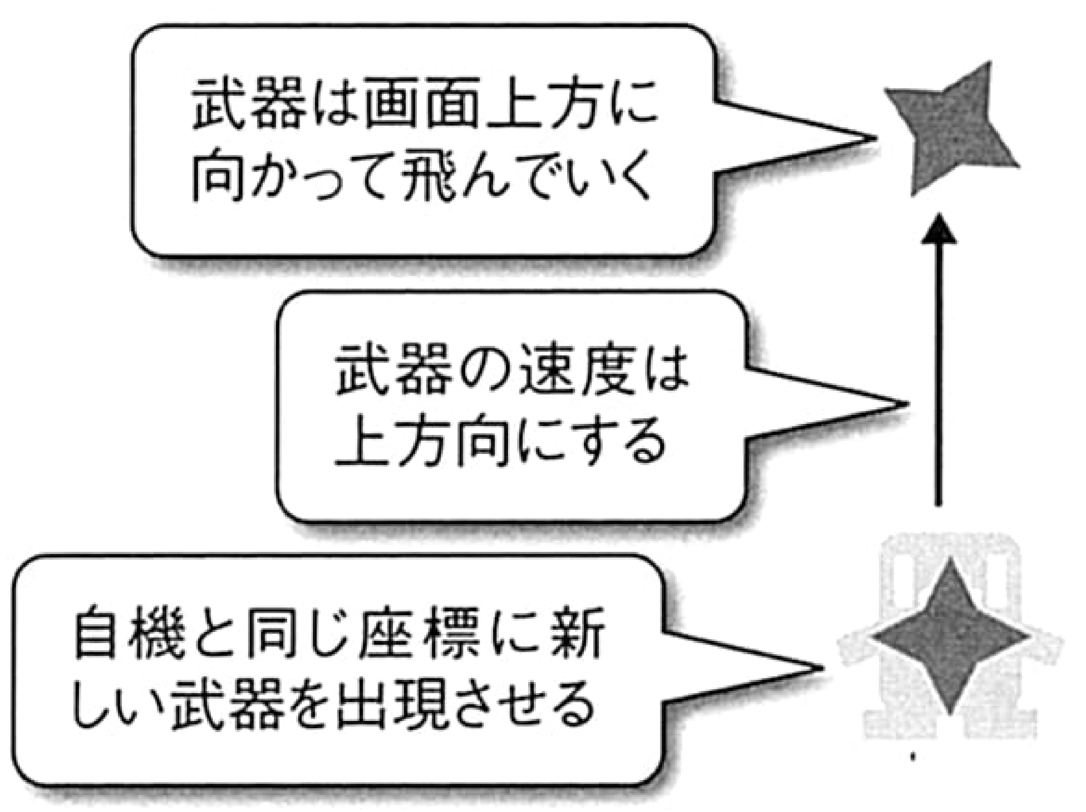
    // アニメーションを再生する
    obj.sprite.stepAnim();

    // アニメーションが終わったらゲームを初期化する
    if (obj.sprite.isAnimOver()) {
      init();
    }
  }
}
```

武器の移動

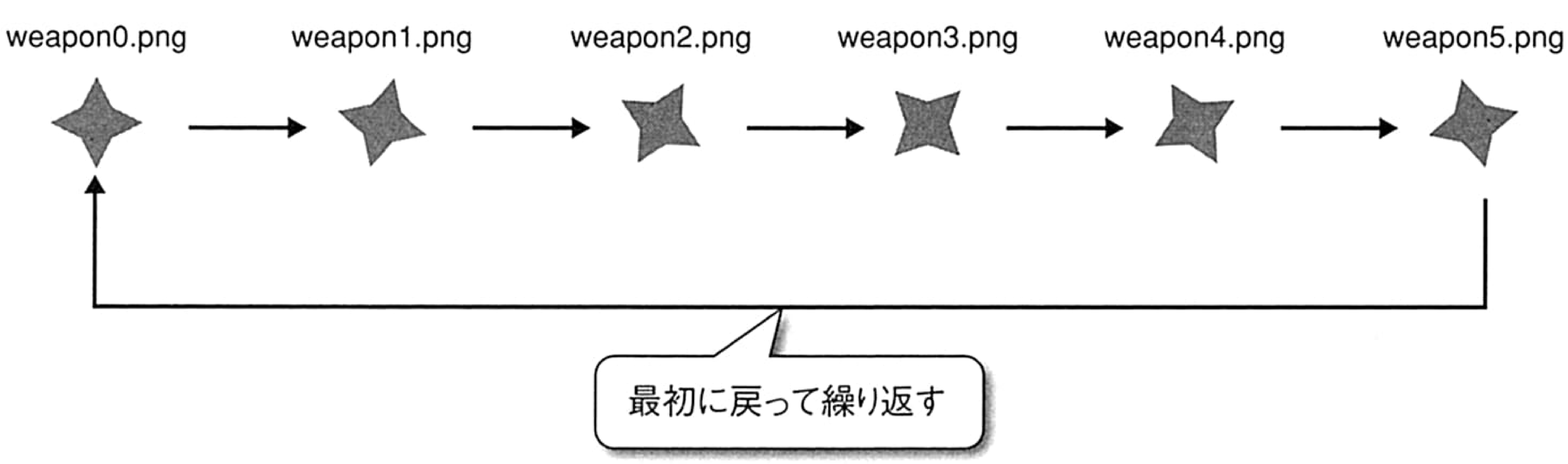
自機が武器を発射したら、武器の座標や速度などを初期化します(Fig.10)。武器の座標は自機と同じにして、速度は画面上方に向かって飛ぶようにします。これで、自機から武器が発射され、上に向かって飛んでいきます。また、武器の状態は通常状態にしておきます。

▼ **Fig.10**: 武器の初期化と移動



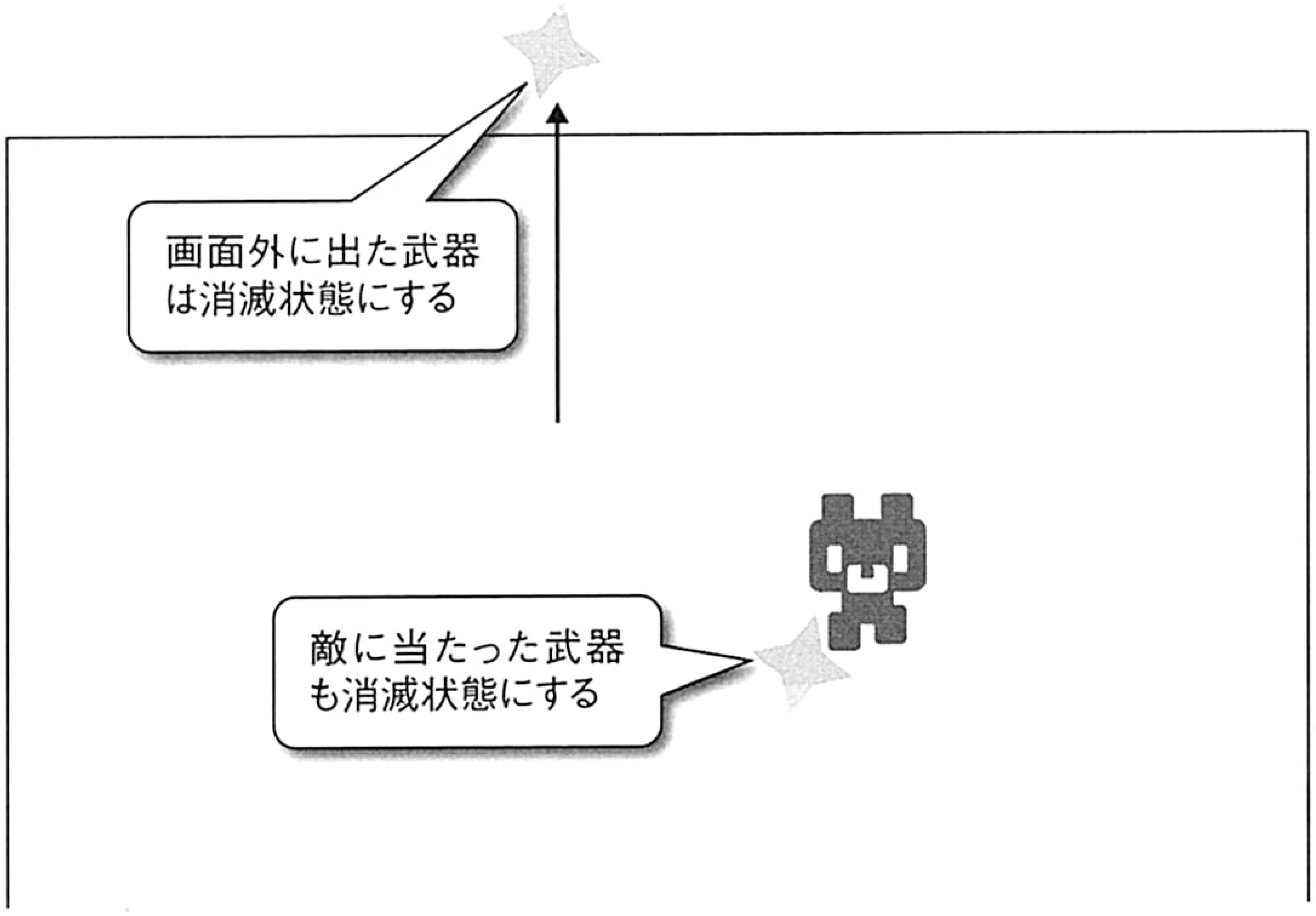
武器は手裏剣状にしてみました。6コマのアニメーションで、回転しながら飛んでいきます(Fig.11)。

▼ **Fig.11**: 武器のアニメーション



武器が画面外に出たら、消滅状態にします(Fig.12)。消滅状態になった武器は、再び初期化して、新しい武器として使うことができます。武器は敵に当たったときにも消滅しますが、詳しくは後述します。

▼ Fig.12: 武器の消滅



武器を移動させるプログラム

List3は武器を動かすプログラムです。武器の処理はWeaponオブジェクトにまとめました。最初は、

```
obj.sprite.setState(STATE_WEAPON_DEAD);
```

のように、武器を消滅状態にしておきます。自機が武器を撃つときには、

```
obj.init=function(x, y, dx, dy) {
    obj.sprite.setXY(x, y);
    obj.vx=dx*WEAPON_SPEED;
    obj.vy=dy*WEAPON_SPEED;
    obj.sprite.setState(STATE_WEAPON_ALIVE);
}
```

のようなinit関数を呼び出して、武器を初期化します。引数を使って武器の座標と速度を設定するとともに、武器を通常状態にします。

一定時間ごとに実行する更新処理(onUpdate関数)では、

```
if (obj.sprite.getState()!=STATE_WEAPON_ALIVE) return;
```

のように、通常状態の武器だけを動かします。消滅状態の武器は再利用されるのを待っているだけなので、動かす必要はありません。

この更新処理では、武器の座標を更新し、アニメーションを進行させます。最後に、

```
if (x<-1 || x>XMAX+1 || y<-1 || y>YMAX+1) {
    obj.sprite.setState(STATE_WEAPON_DEAD);
}
```

として、画面外に出た武器を消滅状態にします。

List3: 武器の移動(shootEnemy.js)

```
// 武器のスピード
var WEAPON_SPEED=0.4;

// 武器の状態(通常・消滅)
var STATE_WEAPON_ALIVE=4;
var STATE_WEAPON_DEAD=5;

// 武器のオブジェクト
function Weapon() {
  var obj=this;

  // スプライト・アニメーション・当たり判定・座標・速度の初期化
  obj.sprite=game.createSprite();
  obj.sprite.initAnim(weaponAnim);
  obj.sprite.initHit(weaponHit);
  obj.sprite.setXY(-1, -1);
  obj.vx=0;
  obj.vy=0;

  // 最初は消滅状態にしておく
  obj.sprite.setState(STATE_WEAPON_DEAD);

  // 武器の初期化(発射)
  obj.init=function(x, y, dx, dy) {

    // 座標と速度の設定
    obj.sprite.setXY(x, y);
    obj.vx=dx*WEAPON_SPEED;
```

```
    obj.vy=dy*WEAPON_SPEED;

    // 通常状態にする
    obj.sprite.setState(STATE_WEAPON_ALIVE);
  }

  // 更新処理
  obj.sprite.onUpdate=function() {

    // 通常状態でなければ処理しない
    if (obj.sprite.getState()!=STATE_WEAPON_ALIVE) return;

    // 座標の更新
    var x=obj.sprite.getX(), y=obj.sprite.getY();
    x+=obj.vx;
    y+=obj.vy;
    obj.sprite.setXY(x, y);

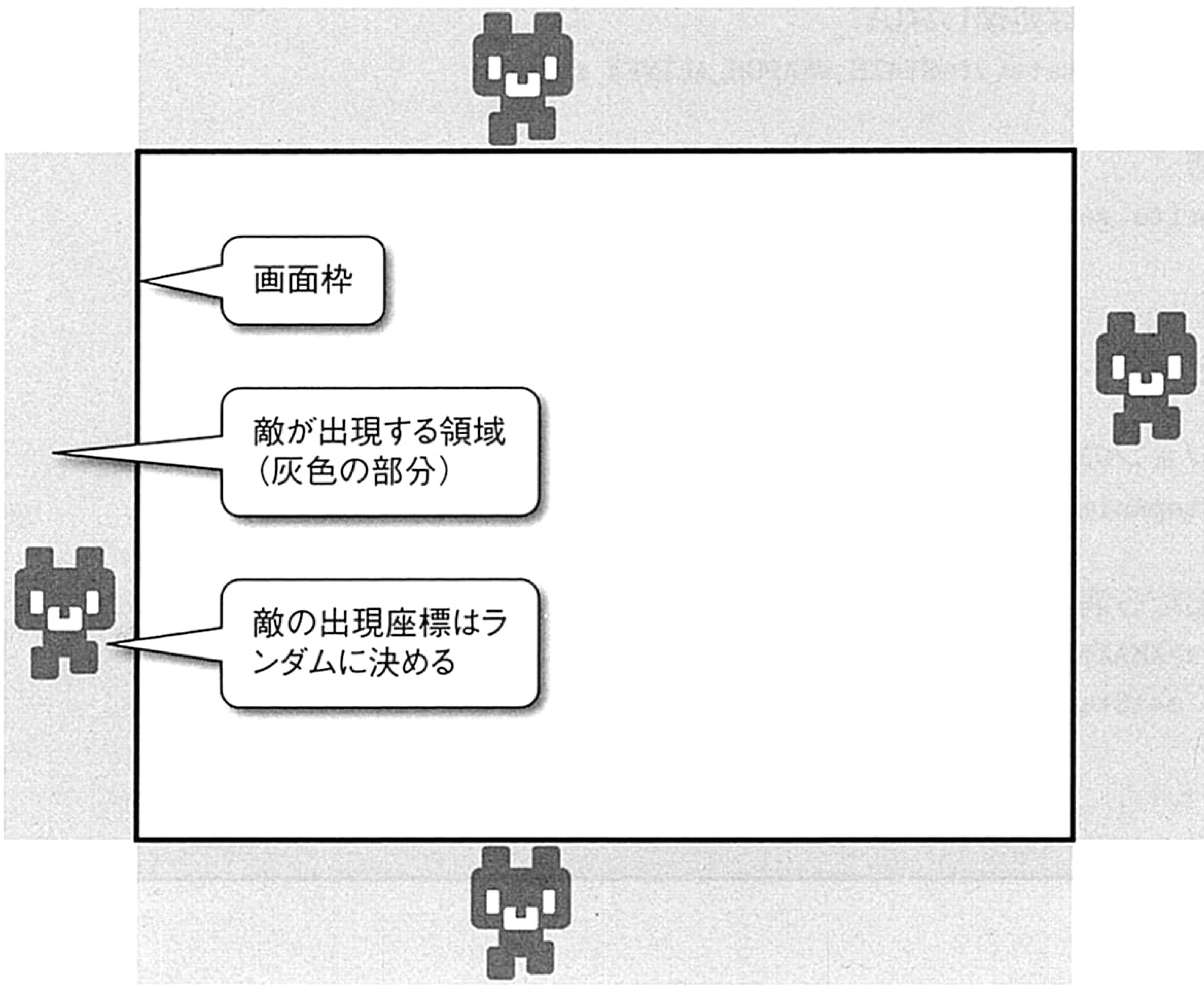
    // アニメーションの進行
    obj.sprite.stepAnim();

    // 画面外に出たら消滅状態にする
    if (x<-1 || x>XMAX+1 || y<-1 || y>YMAX+1) {
      obj.sprite.setState(STATE_WEAPON_DEAD);
    }
  }
}
```

敵を武器で倒す

本章のサンプルに出てくる敵は、第5章の追いかけてくる敵(p.145)とほぼ同じです。出現方法だけは少し変えて、画面枠のすぐ外のランダムな位置に出現させます(Fig.13)。

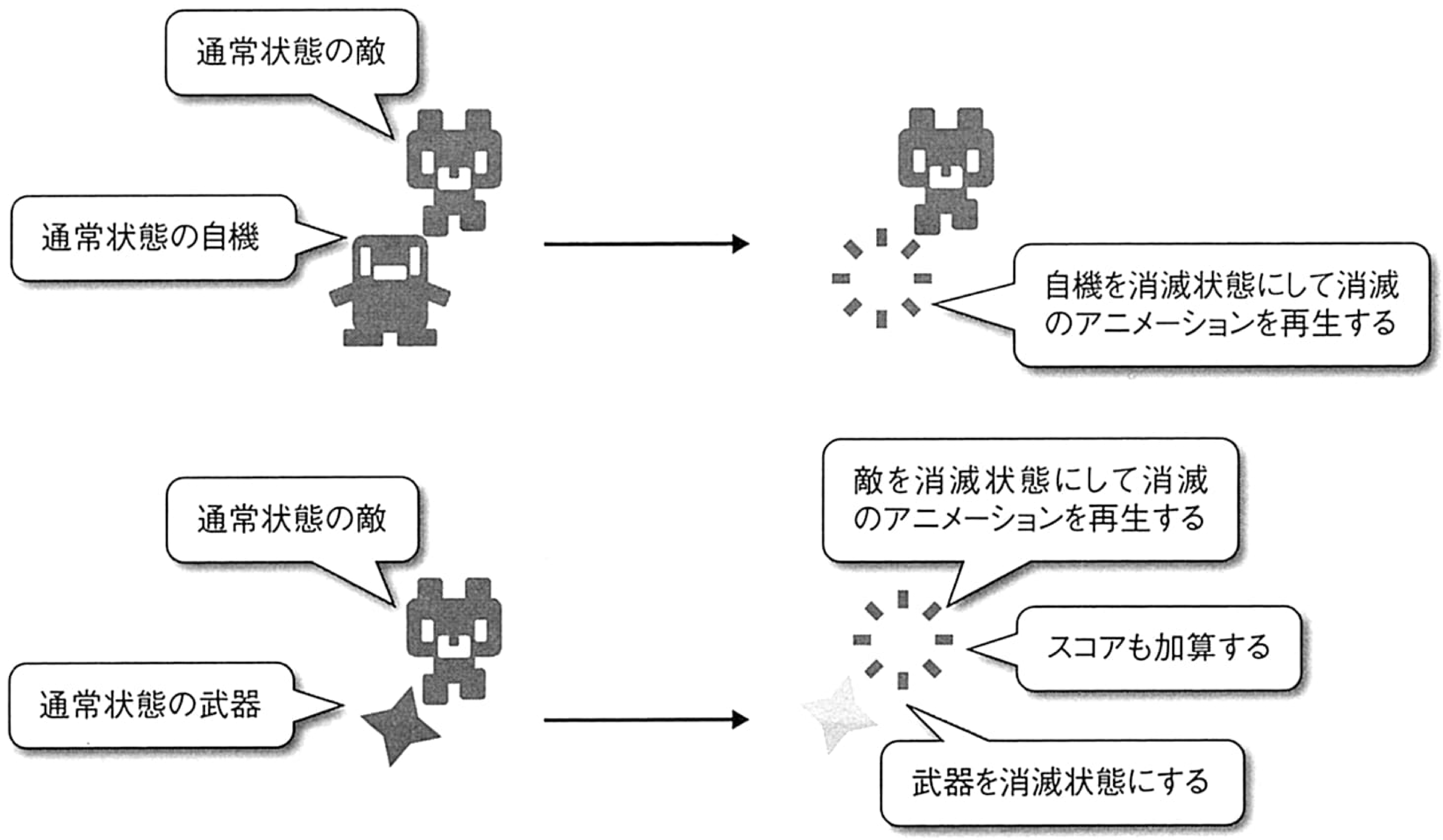
Fig.13: 敵の出現



この敵のポイントは当たり判定処理です(Fig.14)。敵が接触したときに影響があるのは、次の2種類の物体です。

- **通常状態の自機**
  自機を消滅状態にします。自機のアニメーションも消滅のアニメーションに切り替えます。
- **通常状態の武器**
  武器を消滅状態にします。敵も消滅状態にして、消滅のアニメーションに切り替えます。また、スコアを加算します。

▼ Fig.14: 敵の当たり判定処理



このように接触した物体の種類によって異なる処理を行うには、当たり判定処理で物体に接触した際に独自の処理を呼び出す機能(p.94)を使います。独自の処理において、接触した物体の種類を判定し、種類に応じた処理を行います。

武器で倒される敵のプログラム

List4は武器で倒される敵のプログラムです。第5章の追いかけてくる敵のプログラム(p.148のList4)をベースにしました。

敵の初期化はEnemyオブジェクトのinit関数で行います。ここでは、

```
switch (randInt(4)) {
    case 0: obj.sprite.setXY(randFloat2(0, XMAX), -1); break;
    case 1: obj.sprite.setXY(randFloat2(0, XMAX), YMAX+1); break;
    case 2: obj.sprite.setXY(-1, randFloat2(0, YMAX)); break;
    case 3: obj.sprite.setXY(XMAX+1, randFloat2(0, YMAX)); break;
}
```

のようなプログラムで、敵を画面外のランダムな位置に出現させます。まず画面の上下左右いずれかをランダムに選び、次にX座標やY座標をランダムに決めます。

敵には通常状態(STATE_ENEMY_ALIVE)と消滅状態(STATE_ENEMY_DEAD)があります。通常状態の処理はalive関数で、消滅状態の処理はdead関数で行います。

alive関数では、

```
game.testHitSprite(obj.sprite);
```

のように、敵と他のスプライトの当たり判定処理を行います。さらに、

```
obj.sprite.onHitSprite=function(spr) {
```

のようにonHitSprite関数を定義します。この関数は、敵が他のスプライトに接触するたびに呼び出されます。

onHitSprite関数では、

```
switch (spr.getState()) {
```

のようなswitch文を使って、接触したスプライトの種類ごとに異なる処理を行います。

通常状態の自機(STATE_PLAYER_ALIVE)に当たった場合には、

```
spr.initAnim(vanishAnim);
spr.setState(STATE_PLAYER_DEAD);
```

のように、自機に消滅のアニメーションを再生させるとともに、自機を消滅状態にします。

通常状態の武器(STATE_WEAPON_ALIVE)に当たった場合には、

```
spr.setXY(-1, -1);
spr.setState(STATE_WEAPON_DEAD);
```

のように、武器を画面外に出してから、消滅状態にします。画面外に出すのは、消滅状態にするだけでは、その場に武器が残ってしまうからです。また、

```
obj.sprite.initAnim(vanishAnim);
obj.sprite.setState(STATE_ENEMY_DEAD);
```

のように、敵に消滅のアニメーションを再生させるとともに、敵を消滅状態にします。さらに、

```
game.addScore(1);
```

として、スコアに1点を加算します。

▼ List4: 武器で倒される敵(shootEnemy.js)

```
// 敵のスピード
var ENEMY_SPEED=0.1;

// 敵が追跡する確率
var ENEMY_CHASE=0.04;

// 敵が待機する確率
var ENEMY_WAIT=0.02;

// 敵の状態(通常・消滅)
var STATE_ENEMY_ALIVE=2;
var STATE_ENEMY_DEAD=3;

// 敵のオブジェクト
function Enemy() {
  var obj=this;

  // スプライトと当たり判定の初期化
  obj.sprite=game.createSprite();
  obj.sprite.initHit(enemyHit);

  // 敵の初期化
  obj.init=function() {

    // 歩くアニメーションを設定する
    obj.sprite.initAnim(enemyWalkAnim);

    // 画面外のランダムな位置に出現させる
    switch (randInt(4)) {
      case 0: obj.sprite.setXY(randFloat2(0, XMAX), -1); break;
      case 1: obj.sprite.setXY(randFloat2(0, XMAX), YMAX+1); break;
      case 2: obj.sprite.setXY(-1, randFloat2(0, YMAX)); break;
      case 3: obj.sprite.setXY(XMAX+1, randFloat2(0, YMAX)); break;
    }

    // 通常状態にする
    obj.sprite.setState(STATE_ENEMY_ALIVE);

    // 追跡する
    obj.chase=true;
  }
  obj.init();

  // 更新処理
```

```
obj.sprite.onUpdate=function() {

  // 状態に応じて分岐する
  switch (obj.sprite.getState()) {
    case STATE_ENEMY_ALIVE: obj.alive(); break;
    case STATE_ENEMY_DEAD: obj.dead(); break;
  }
}

// 通常状態
obj.alive=function() {

  // 敵と自機の座標を取得する
  var x=obj.sprite.getX(), y=obj.sprite.getY();
  var px=player.sprite.getX(), py=player.sprite.getY();

  // 追跡するならば自機を追いかける
  if (obj.chase) {
    if (Math.abs(px-x)<ENEMY_SPEED) {
      x=px;
    } else {
      if (px<x) x-=ENEMY_SPEED; else x+=ENEMY_SPEED;
    }
    if (Math.abs(py-y)<ENEMY_SPEED) {
      y=py;
    } else {
      if (py<y) y-=ENEMY_SPEED; else y+=ENEMY_SPEED;
    }
    if (randFloat(1)<ENEMY_WAIT) obj.chase=false;
  } else

  // 待機するならばその場で待つ
  {
    if (randFloat(1)<ENEMY_CHASE) obj.chase=true;
  }

  // 移動中はアニメーションを再生する
  if (x!=obj.sprite.getX() || y!=obj.sprite.getY()) {
    obj.sprite.stepAnim();
  }

  // 座標の更新
  obj.sprite.setXY(x, y);

  // 当たり判定処理
  game.testHitSprite(obj.sprite);
}
```

```
  // 他のスプライトに接触したときに呼び出される関数
  obj.sprite.onHitSprite=function(spr) {

    // 他のスプライトの状態に応じて分岐する
    switch (spr.getState()) {

      // 通常状態の自機に接触したとき
      case STATE_PLAYER_ALIVE:

        // 自機に消滅のアニメーションを再生させるとともに、
        // 消滅状態にする
        spr.initAnim(vanishAnim);
        spr.setState(STATE_PLAYER_DEAD);
        break;

      // 通常状態の武器に接触したとき
      case STATE_WEAPON_ALIVE:

        // 武器を画面外に出して見えなくする
        spr.setXY(-1, -1);

        // 武器を消滅状態にする
        spr.setState(STATE_WEAPON_DEAD);

        // 敵に消滅のアニメーションを再生させるとともに、
        // 消滅状態にする
        obj.sprite.initAnim(vanishAnim);
        obj.sprite.setState(STATE_ENEMY_DEAD);

        // スコアを加算する
        game.addScore(1);
        break;
    }
  }

  // 消滅状態
  obj.dead=function() {

    // アニメーションを再生する
    obj.sprite.stepAnim();

    // アニメーションが終わったらゲームを初期化する
    if (obj.sprite.isAnimOver()) {
      obj.init();
    }
  }
}
```

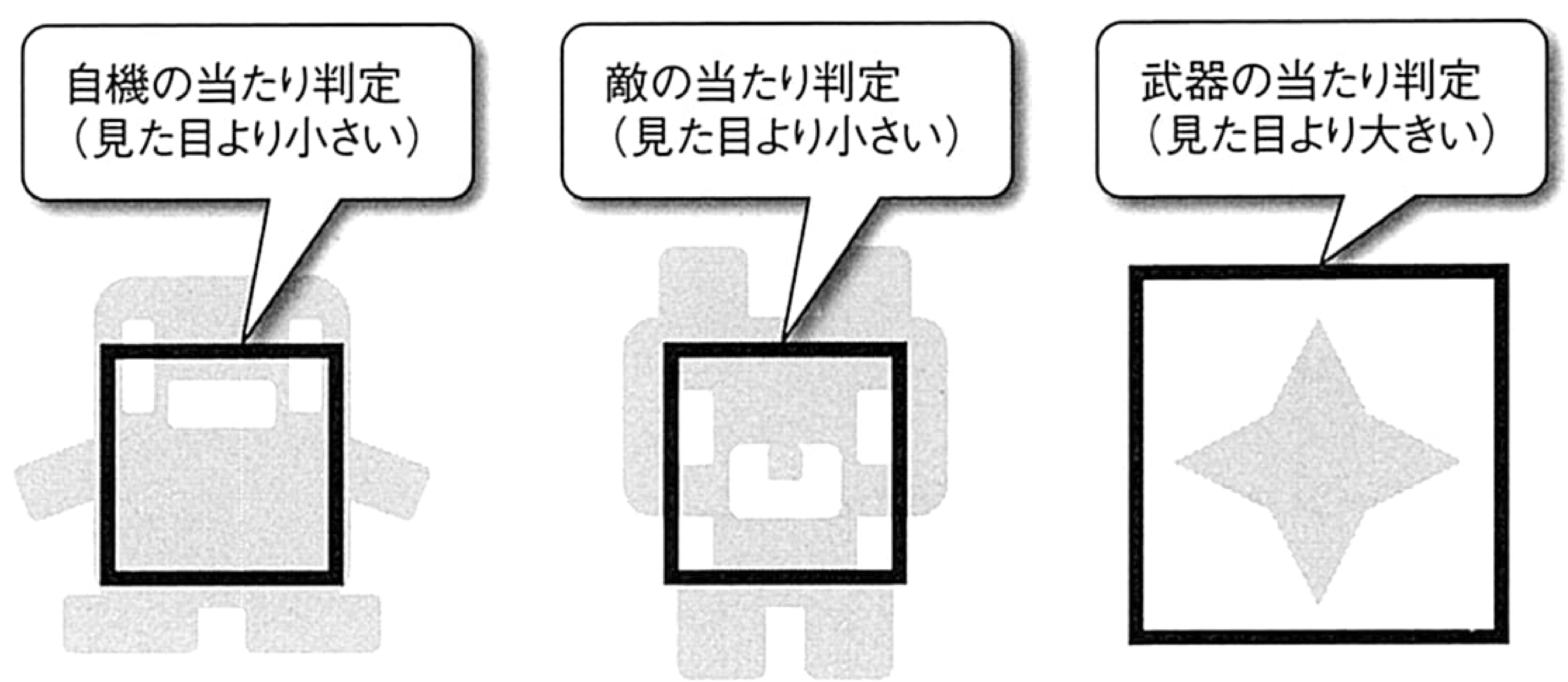
武器で敵を撃つゲームの本体

List5は武器で敵を撃つゲームの本体です。これまでのサンプルと同じように、ゲームの初期化を行うinit関数と、メインルーチンのmain関数を定義します。

init関数では、画面を消去し、スコアを0にして、自機・敵・武器のオブジェクトを作成します。main関数では、アニメーションや当たり判定の初期化を行います。

自機や敵の当たり判定は見た目よりも小さめですが、武器の当たり判定は逆に大きめにしています(Fig.15)。これは敵に武器を当てやすくするためです。武器が当たったように見えたのに当たらない、というのはプレイヤーにとってストレスになるので、武器はある程度当たりやすくしておくのがよいでしょう。

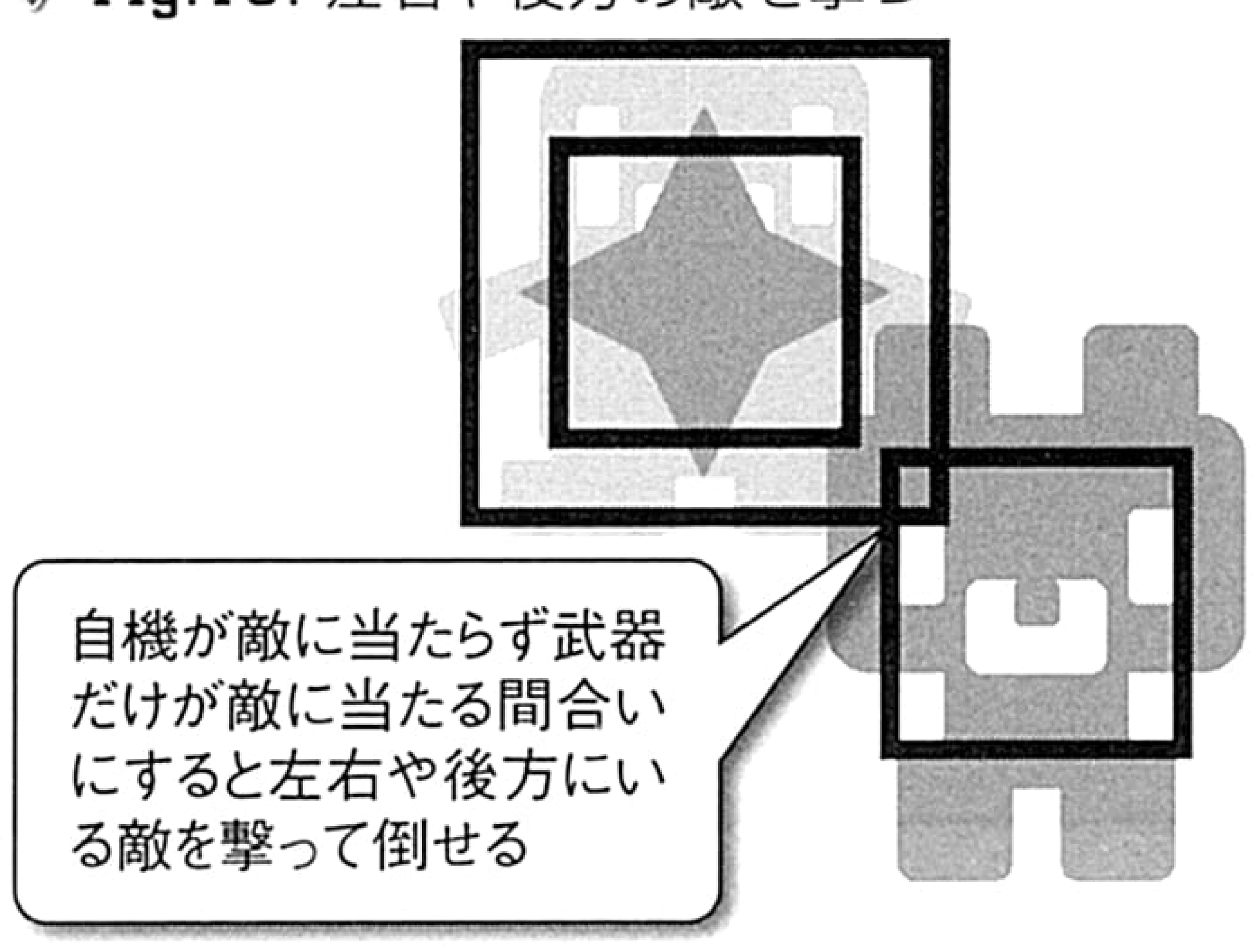
Fig.15: 武器の当たり判定



自機の当たり判定を小さく、武器の当たり判定を大きくしておくと、武器を撃ったときに面白い現象が起きます(Fig.16)。自機は敵に当たらず、武器は敵に当たるという絶妙の間合いで武器を撃てば、前方だけではなく、左右や後方にいる敵も倒すことができます。

これは見た目が面白く、実用性もあるテクニックですが、好き嫌いが分かれるかもしれません。前方の敵だけを撃ちたいときには、武器が出る位置を自機の前方に少しずらすとよいでしょう。

Fig.16: 左右や後方の敵を撃つ



▼ **List5**: ゲーム本体(shootEnemy.js)

```
// 武器の数
var WEAPON_COUNT=5;

// 敵の数
var ENEMY_COUNT=10;

// ゲーム本体・自機・敵・武器のオブジェクト
var game;
var player;
var enemy;
var weapon;

// 自機・敵・消滅・武器のアニメーション
var playerWalkAnim;
var enemyWalkAnim;
var vanishAnim;
var weaponAnim;

// 自機・敵・武器の当たり判定
var playerHit;
var enemyHit;
var weaponHit;

// ゲームの初期化
function init() {
  var i;

  // 画面の消去
  game.clear();

  // スコアの初期化
  game.setScore(0);

  // 自機の作成
  player=new Player();

  // 敵の作成
  enemy=new Array();
  for (i=0; i<ENEMY_COUNT; i++) {
    enemy[i]=new Enemy();
  }

  // 武器の作成
  weapon=new Array();
  for (i=0; i<WEAPON_COUNT; i++) {
    weapon[i]=new Weapon();
  }
```

```
}

// メインルーチン
function main() {

  // ゲーム本体の作成
  game=new Game();

  // アニメーションの作成
  playerWalkAnim=new Array(
    "pcWalk0.png", "pcWalk2.png", "pcWalk0.png",
    "pcWalk1.png", "pcWalk3.png", "pcWalk1.png"
  );
  enemyWalkAnim=new Array(
    "enemyWalk0.png", "enemyWalk2.png", "enemyWalk0.png",
    "enemyWalk1.png", "enemyWalk3.png", "enemyWalk1.png"
  );
  vanishAnim=new Array(
    "void.png", "vanish8.png", "vanish6.png",
    "vanish4.png", "vanish2.png", "vanish0.png",
    "vanish1.png", "vanish2.png", "vanish3.png",
    "vanish4.png", "vanish5.png", "vanish6.png",
    "vanish7.png", "vanish8.png", "void.png"
  );
  weaponAnim=new Array(
    "weapon0.png", "weapon1.png", "weapon2.png",
    "weapon3.png", "weapon4.png", "weapon5.png"
  );

  // 当たり判定の作成
  playerHit=new Array(0.25, 0.25, 0.75, 0.75);
  enemyHit=new Array(0.25, 0.25, 0.75, 0.75);
  weaponHit=new Array(0.1, 0.1, 0.9, 0.9);

  // ゲームの初期化
  init();
}
```

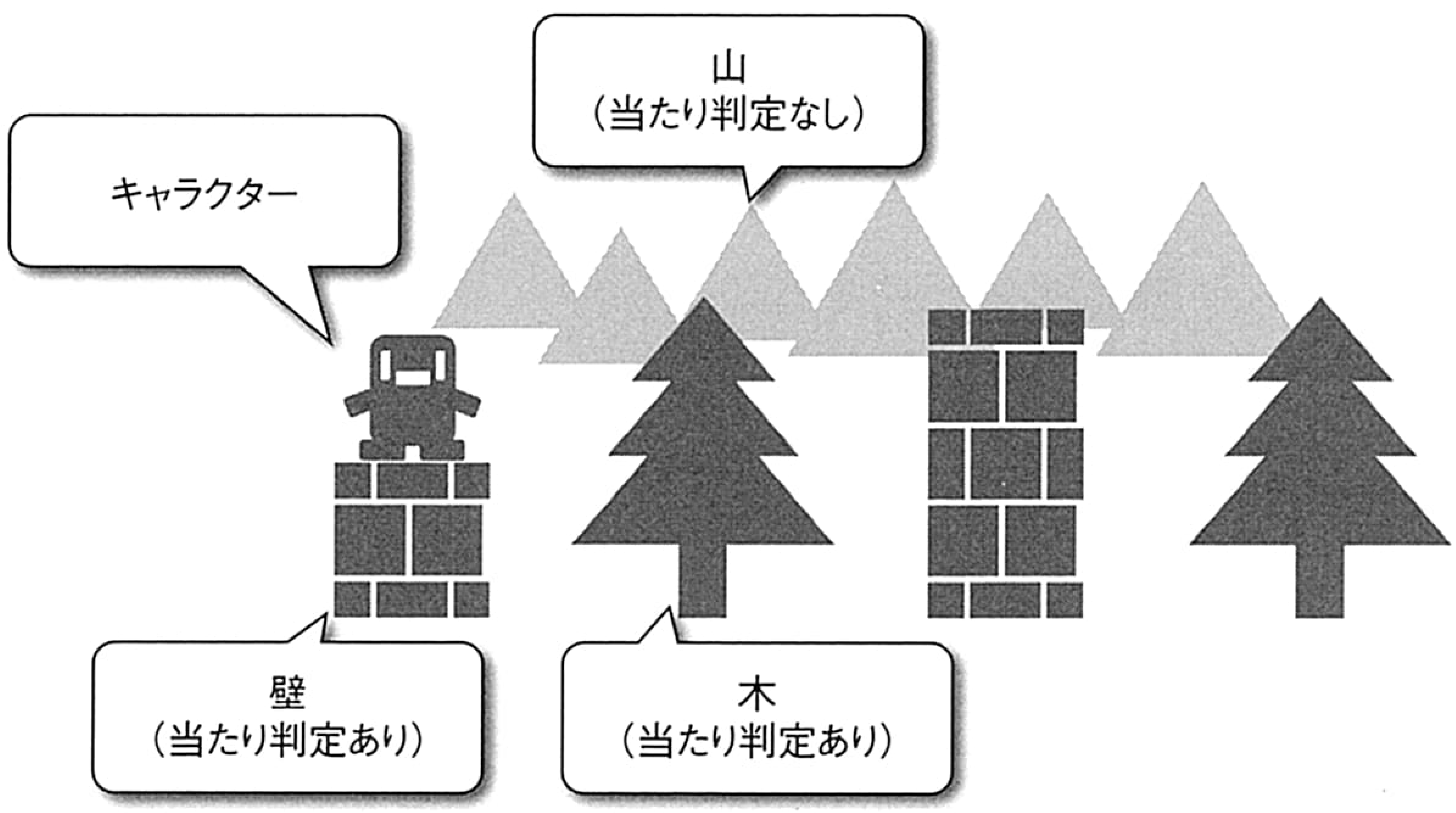
Chapter 7

フィールド

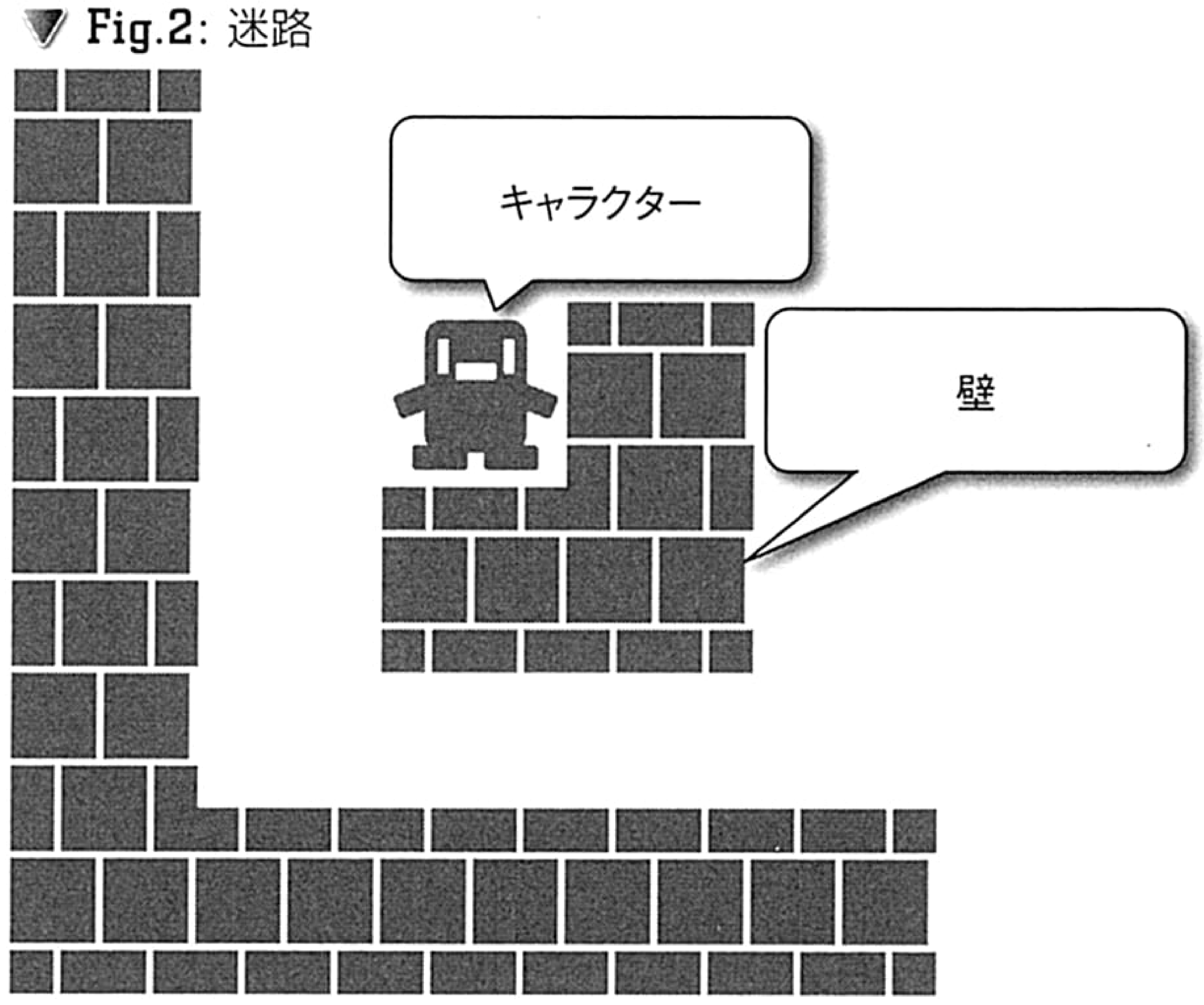
フィールドとスクロール

多くのゲームには舞台があります。舞台は山や川といった自然の風景だったり(Fig.1)、迷路や要塞といった人工の地形だったりします(Fig.2)。また、単に画面をにぎやかにするために表示する舞台もあれば、キャラクターの障害物や足場になる舞台もあります(Fig.3)。本書では、こういったゲームの舞台をフィールドと呼ぶことにします。マップ、ステージ、レベルといった呼び方もあります。

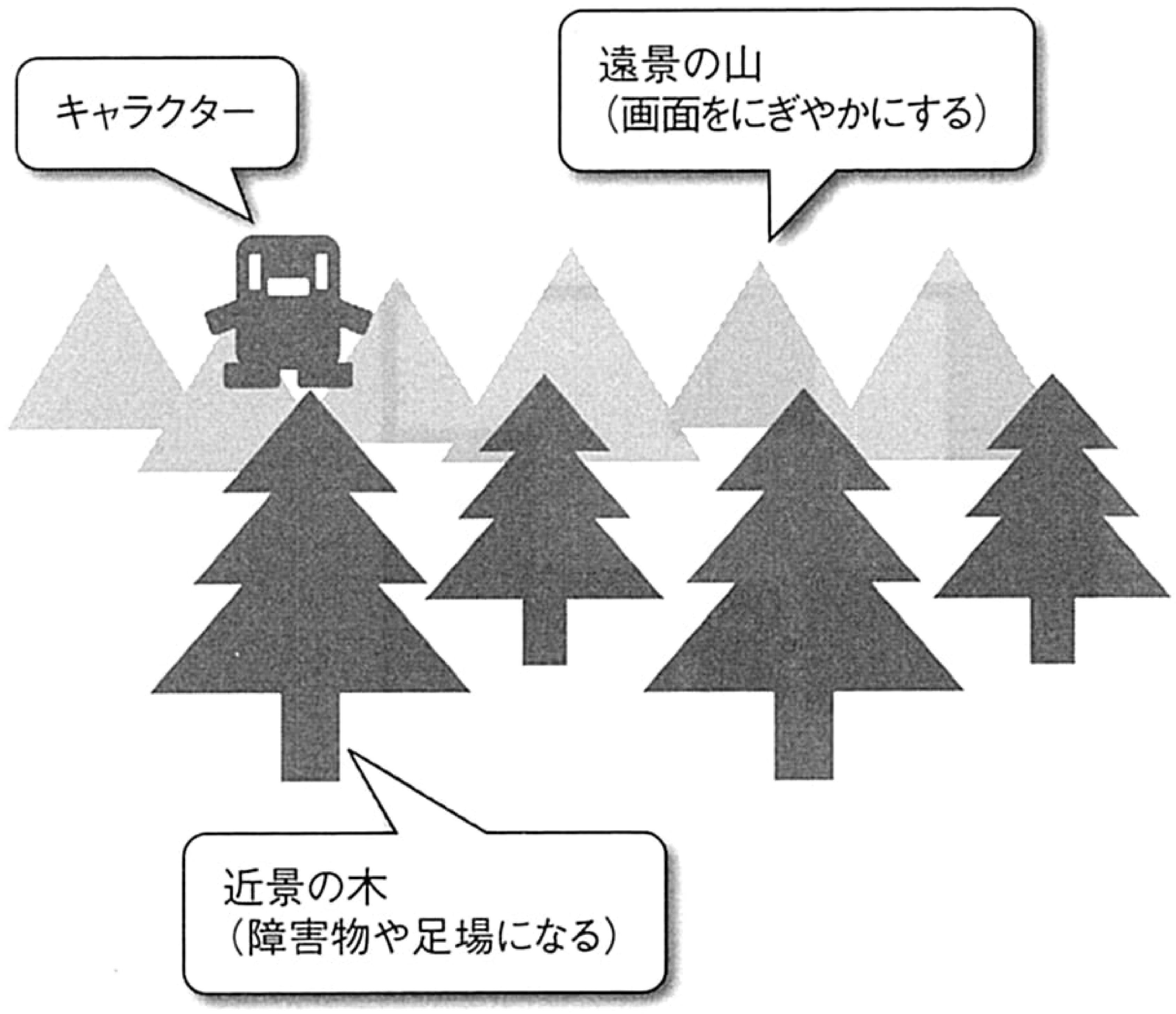
▼ **Fig.1**: 自然の風景



▼ **Fig.2**: 迷路

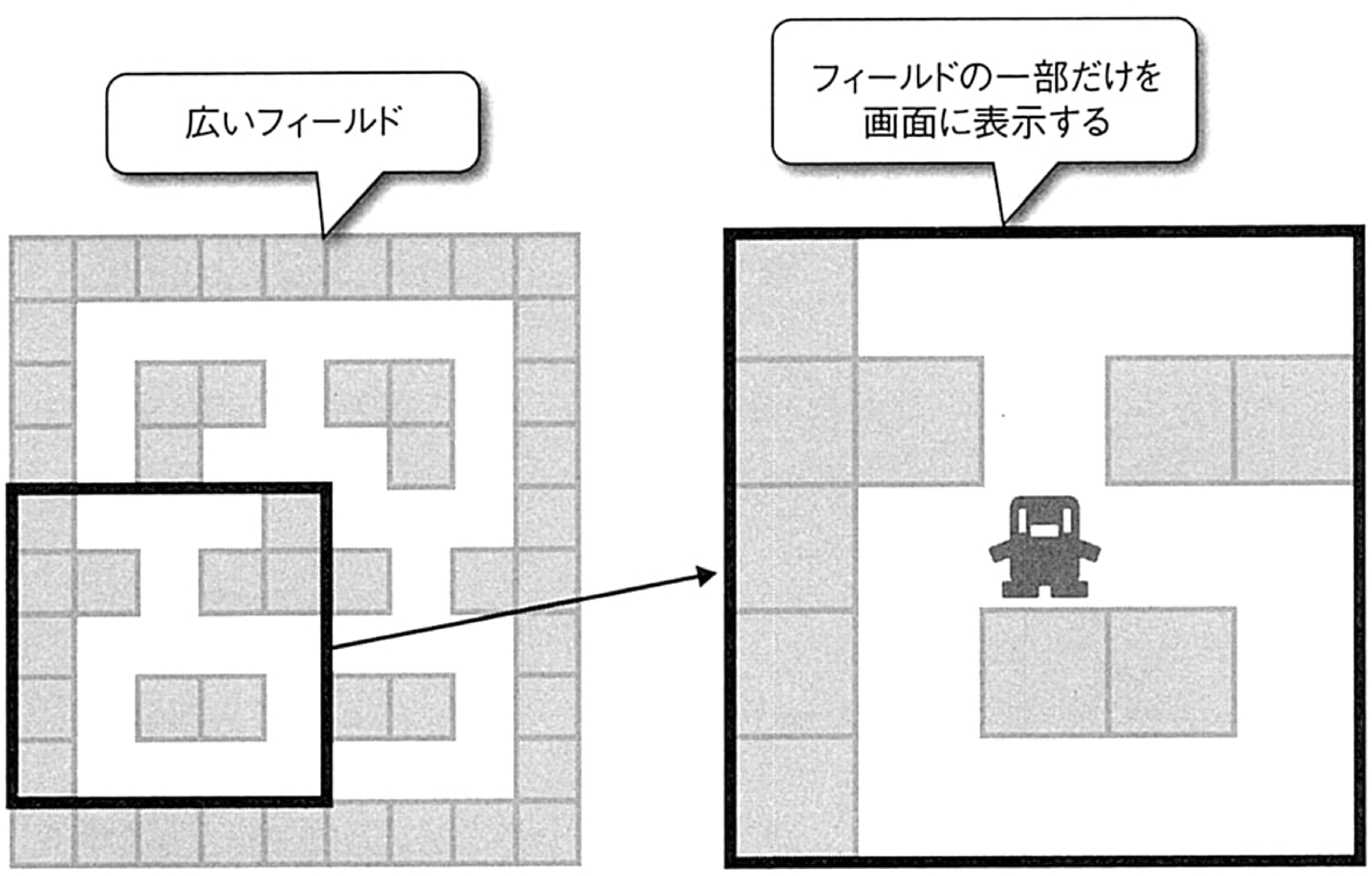


▼ **Fig.3**: キャラクターと舞台の関係

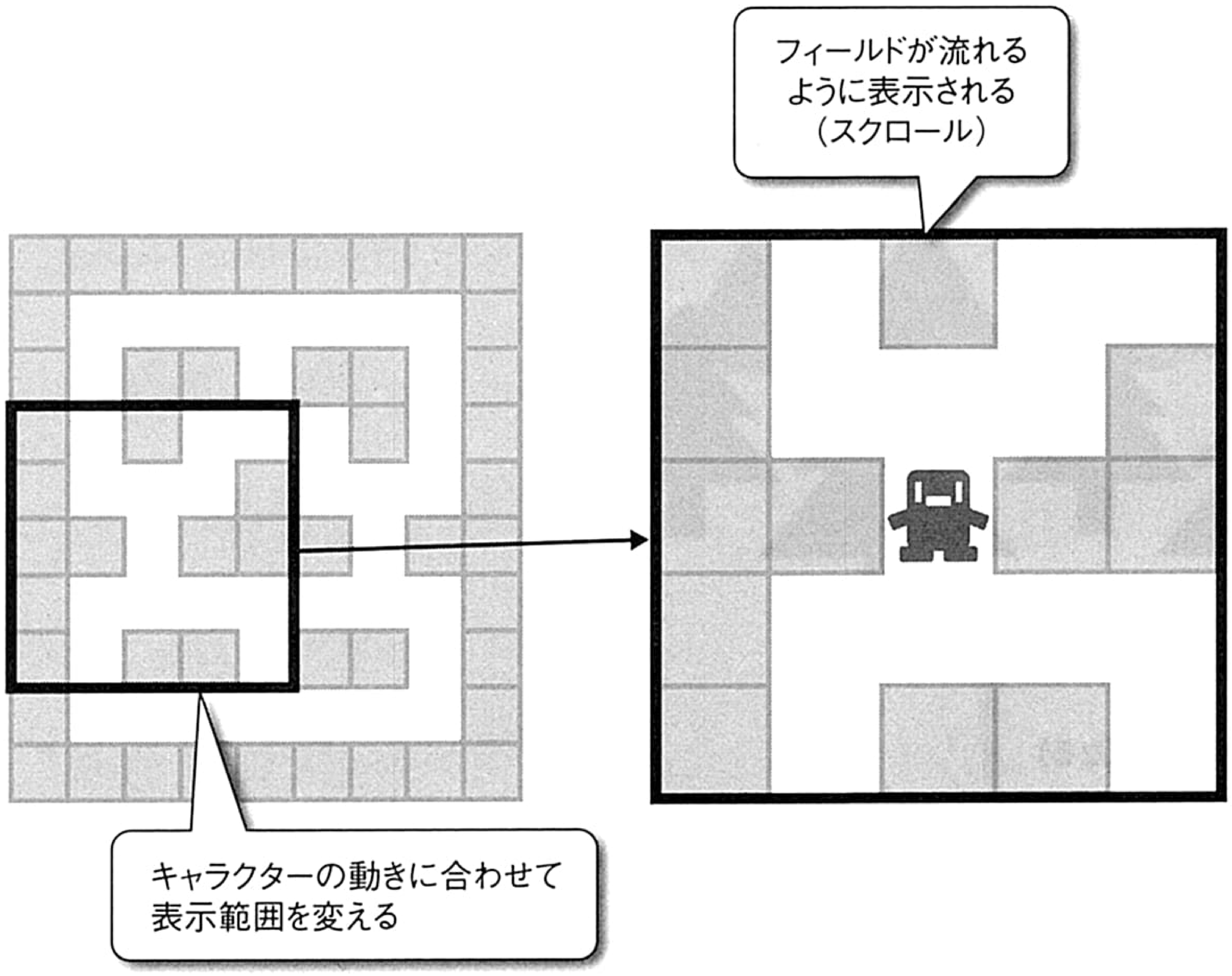


ゲームによってはフィールドが広大で、画面内に一度では表示しきれないことがあります(Fig.4)。こういったゲームでは、キャラクターが動き回るのにあわせて、フィールドの表示範囲を変化させます(Fig.5)。画面上では、キャラクターの進行方向に応じてフィールドが流れるように表示されます。これをスクロールと呼びます。

▼ **Fig.4**: 広大なフィールド



▼ Fig.5: フィールドの表示範囲を変える



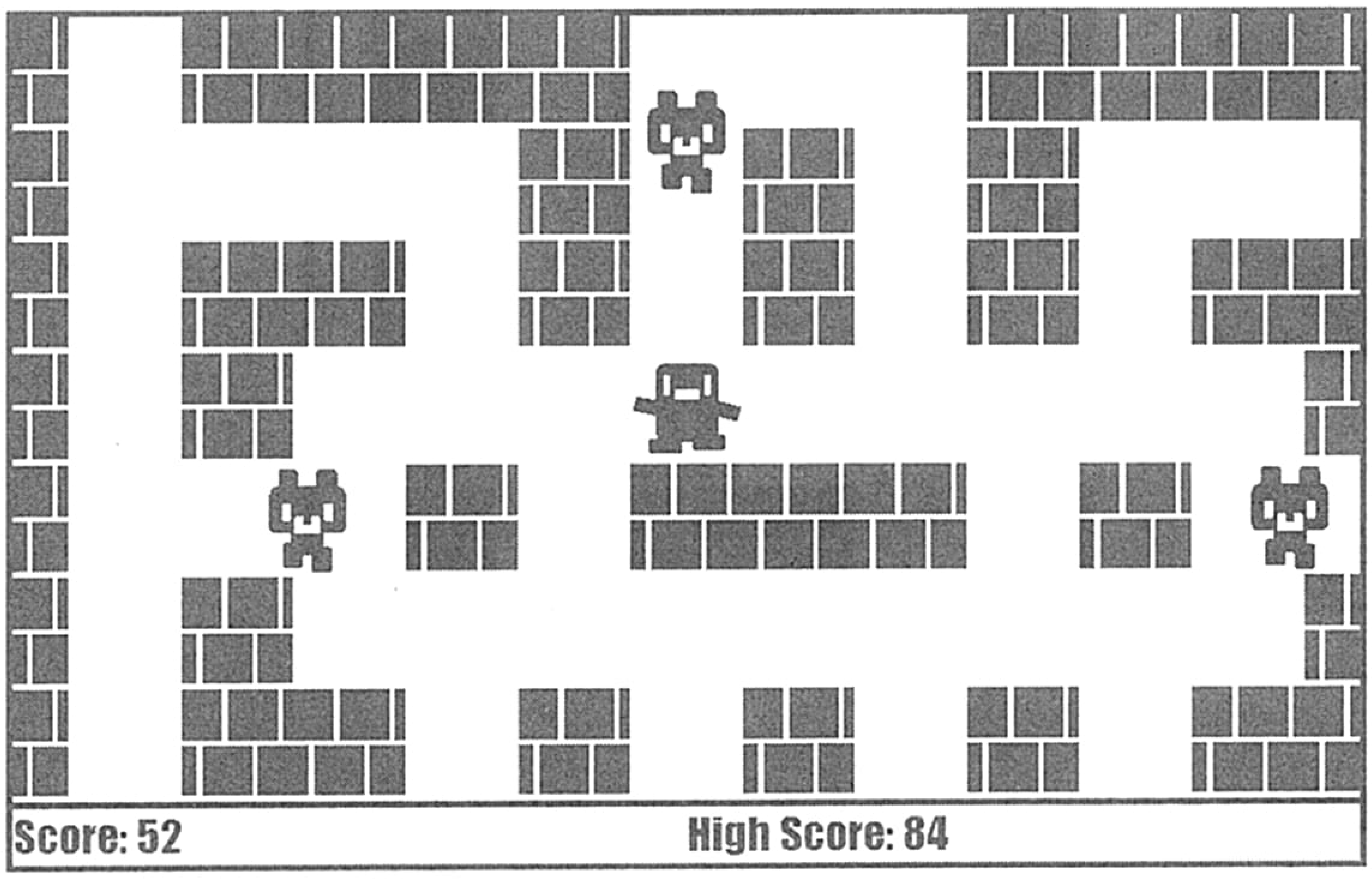
スクロール

画面に表示しきれないテキストや画像などを、水平・垂直・斜めにスライドさせながら表示する手法です。ゲームでは広いフィールドの中を動き回ることが多いので、スクロールをよく利用します。
プレイヤーの操作に関係なく、決まった方向と速度で自動的にスクロールすることを強制スクロールと呼びます。一方、プレイヤーの操作によって方向や速度を制御できるスクロールのことを、任意スクロールや自由スクロールと呼ぶことがあります。スクロールの方式によって、ゲーム性は大きく変わります。

スクロールするフィールドで敵から逃げる

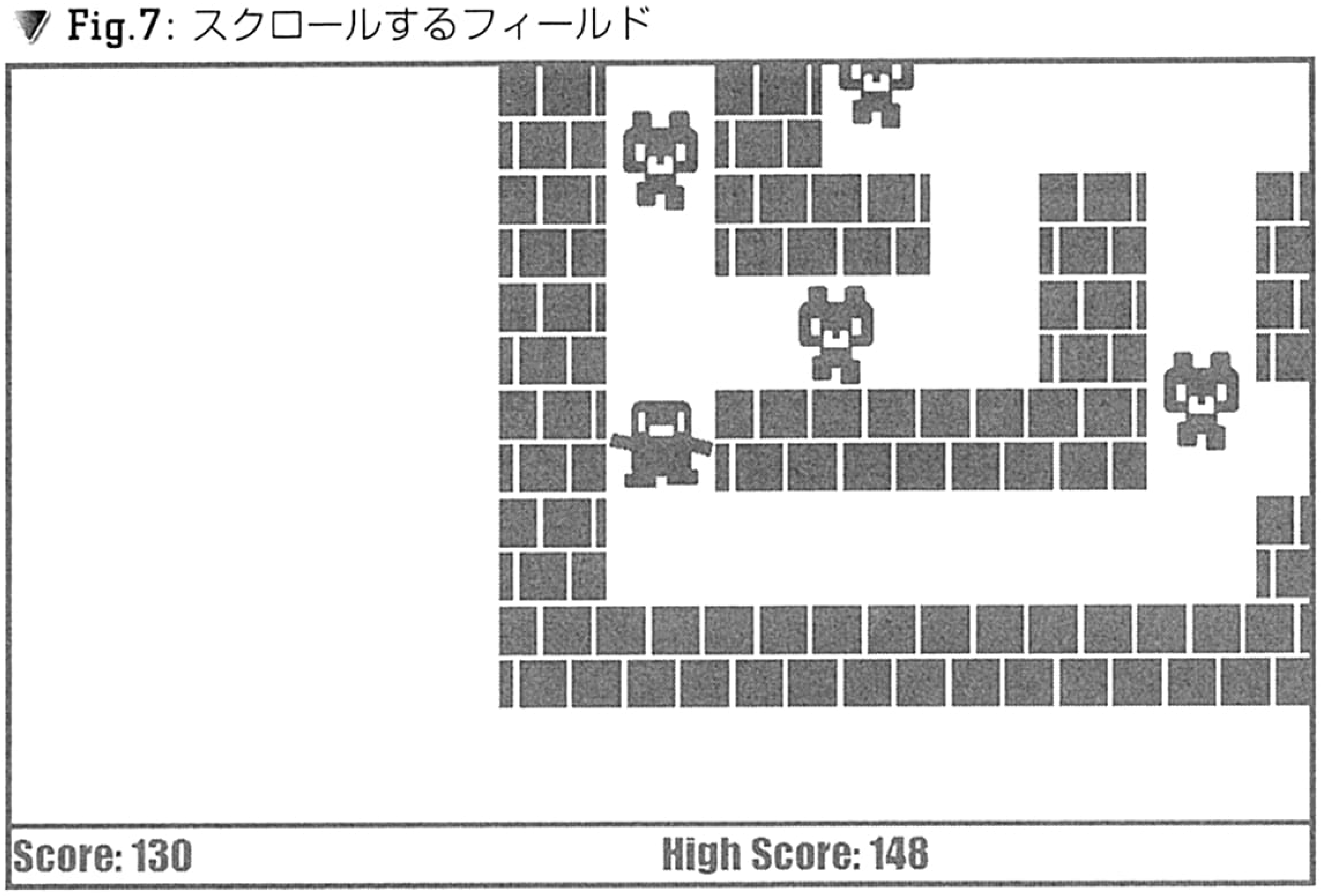
本章のサンプルは、スクロールするフィールドを歩き回り、追いかけてくる敵から逃げるゲームです(Fig.6)。ゲームを実行するには、mazeChase.htmlをブラウザで開きます。

Fig.6: 追いかけてくる敵



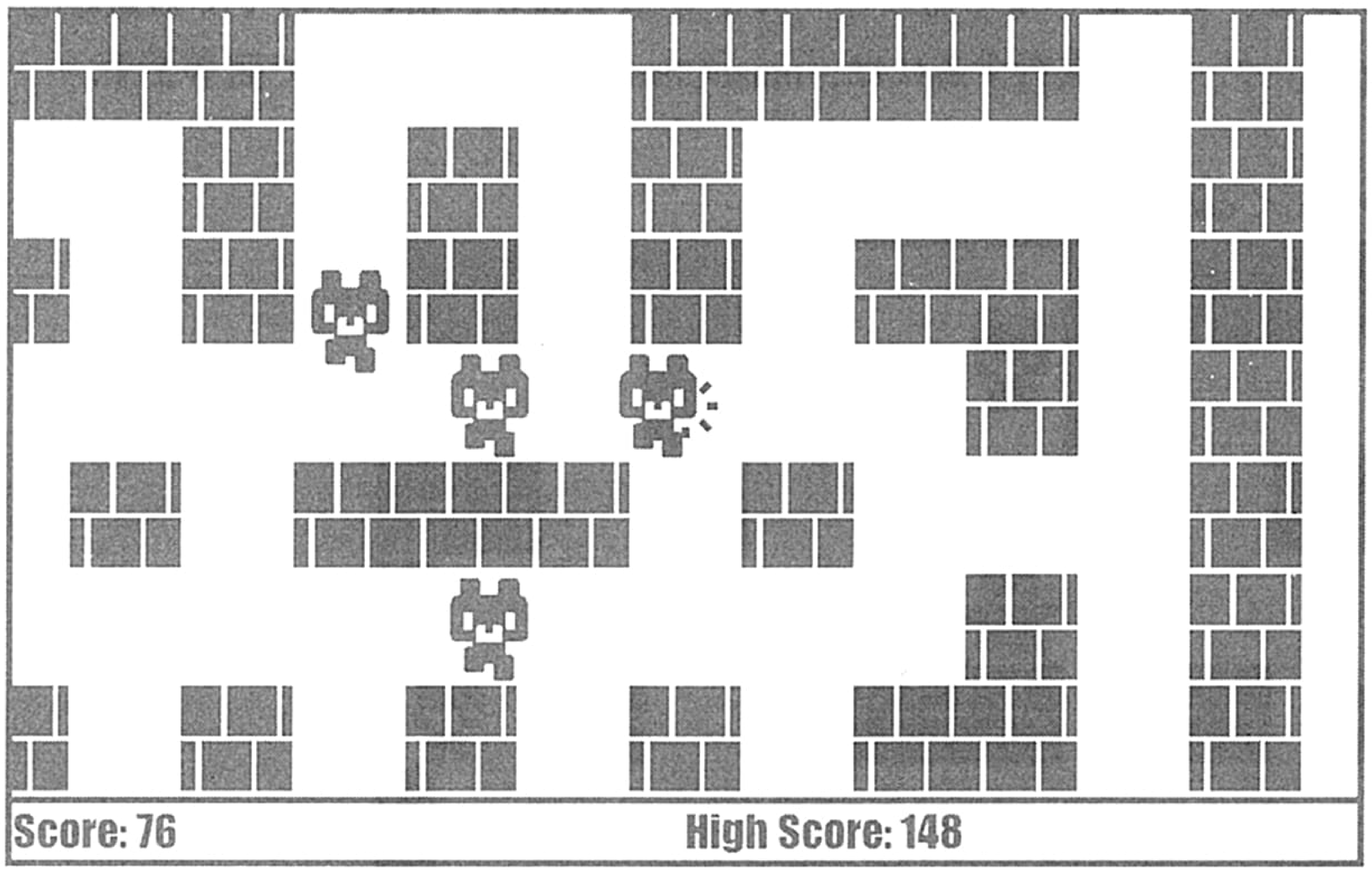
自機は画面の中央に出現します。ゲームが始まると、周囲から敵が迫ってきます。カーソルキーの上下左右で自機を動かして、敵から逃げ回ってください。フィールドは自機の動きに合わせて上下左右にスクロールします。

Fig.7: スクロールするフィールド



敵につかまると、自機が消滅してゲームオーバーです(Fig.8)。長く逃げ回るほど、高いスコアが得られます。

▼ Fig.8: 敵につかまる



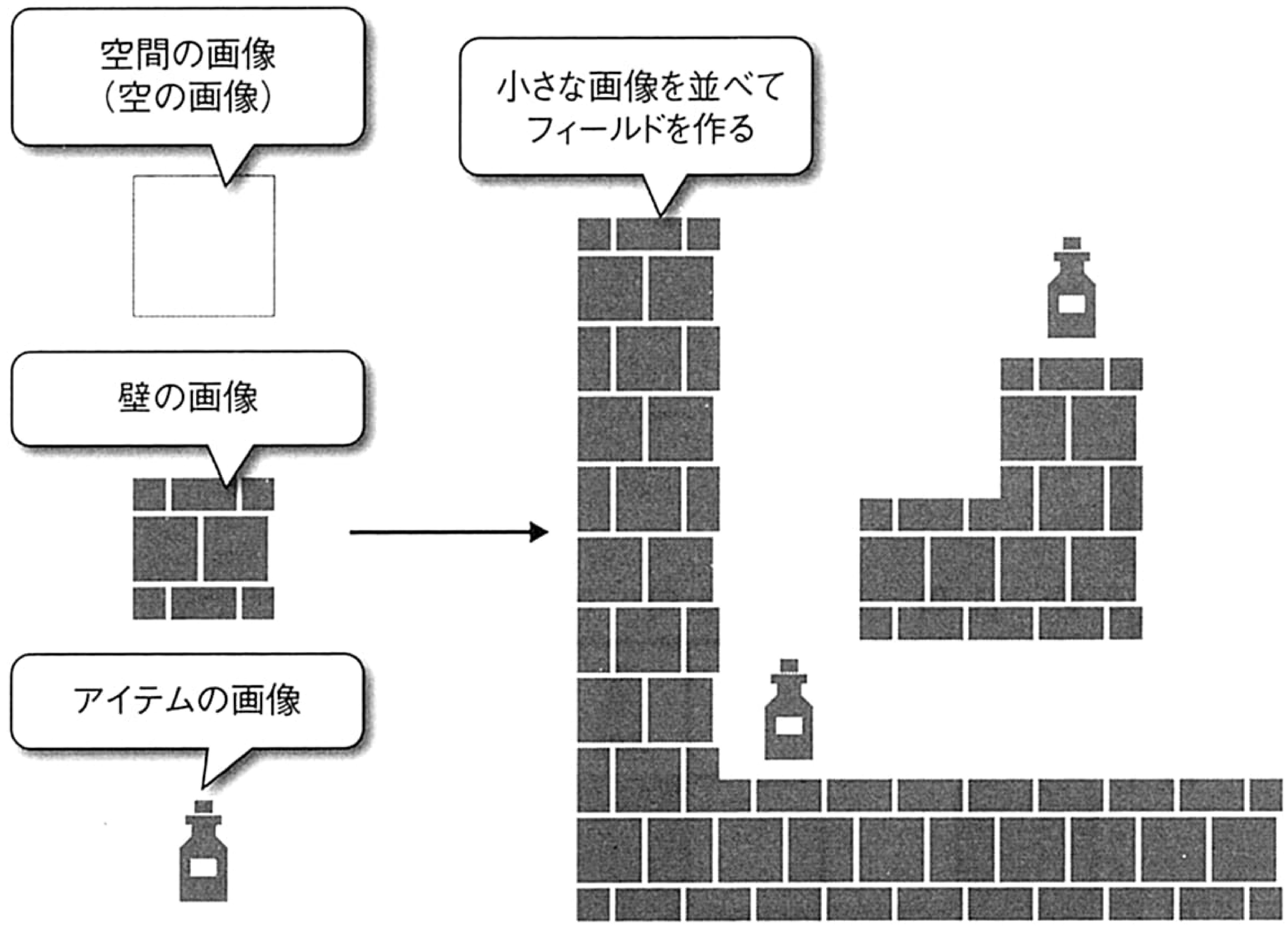
JavaScriptの実行速度はブラウザによって多少違います。特にこのサンプルでは速度の差が顕著に感じられるようです。もしお使いのブラウザで速度が極端に遅かったり、表示が乱れたりすることがあったら、ぜひ別のブラウザで実行してみてください。なお、本書のサンプルはInternet Explorer, Opera, Firefox, Safari, Chromeに対応しています(p.26)。

フィールドの管理

フィールドは一枚の大きな画像に似ていますが、単純に画像として表現してしまうと、データが大きくなりすぎるという問題があります。巨大な画像を描くのは大変ですし、キャラクターとの間で当たり判定処理を行うには、当たり判定を設定する方法も考える必要があります。

こういった問題を回避するためによく知られているのは、小さな画像を縦横に並べてフィールドを表現する方法です(Fig.9)。この方法ならば、壁やアイテムといった同じ画像を何度も使い回せるので、データは小さくなります。

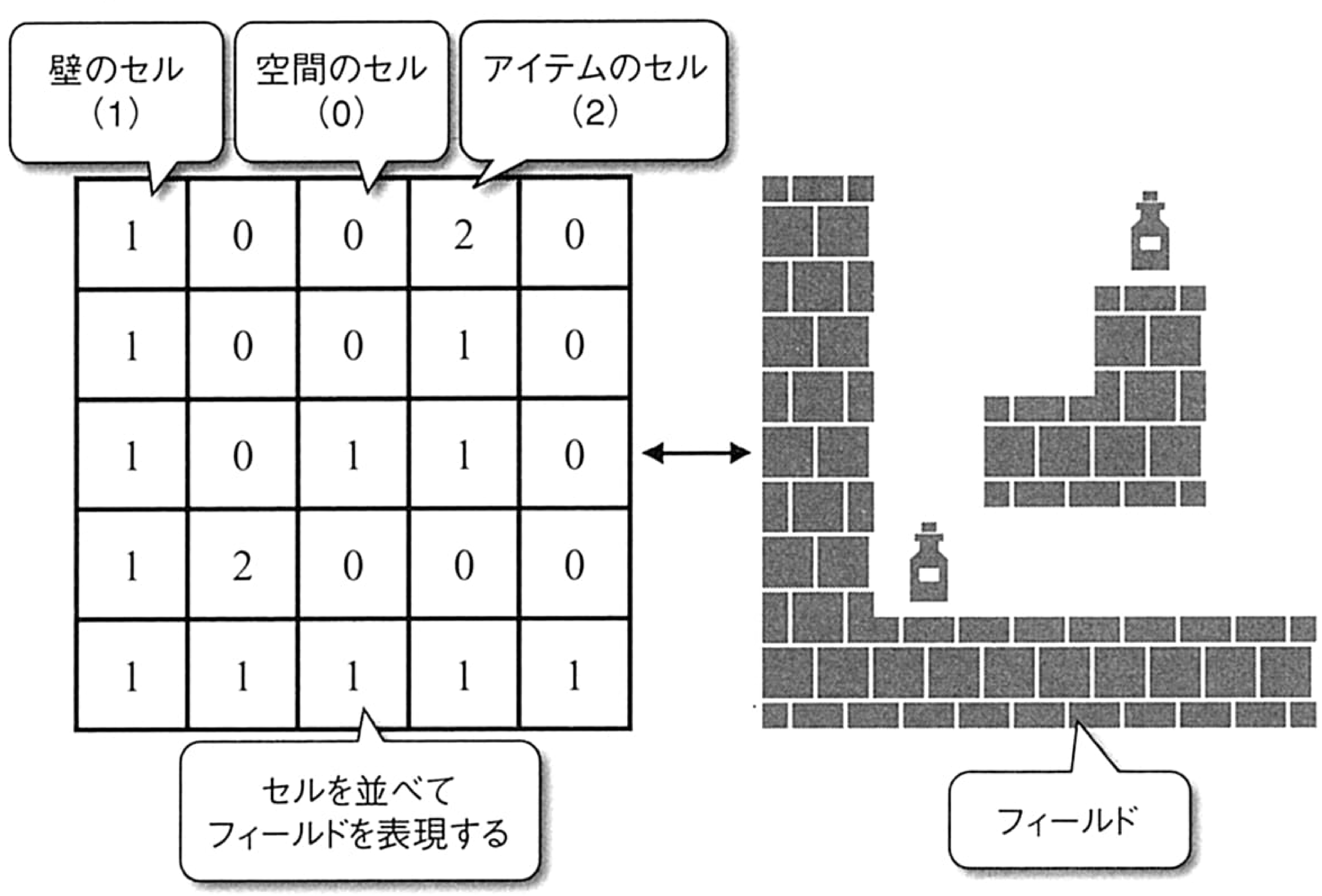
▼ **Fig.9**: 小さな画像を並べてフィールドを表現する



フィールドを構成する壁やアイテムといった要素を、本書ではセルと呼ぶことにします。チップやブロックなどと呼ぶこともあります。

セルには種類に応じて番号を割り振っておきます(Fig.10)。空間は0番、壁は1番、アイテムは2番といった要領です。フィールド全体はセルを縦横に並べたものとして表現できます。プログラムでは、セルの種類を2次元配列などで管理するとよいでしょう。

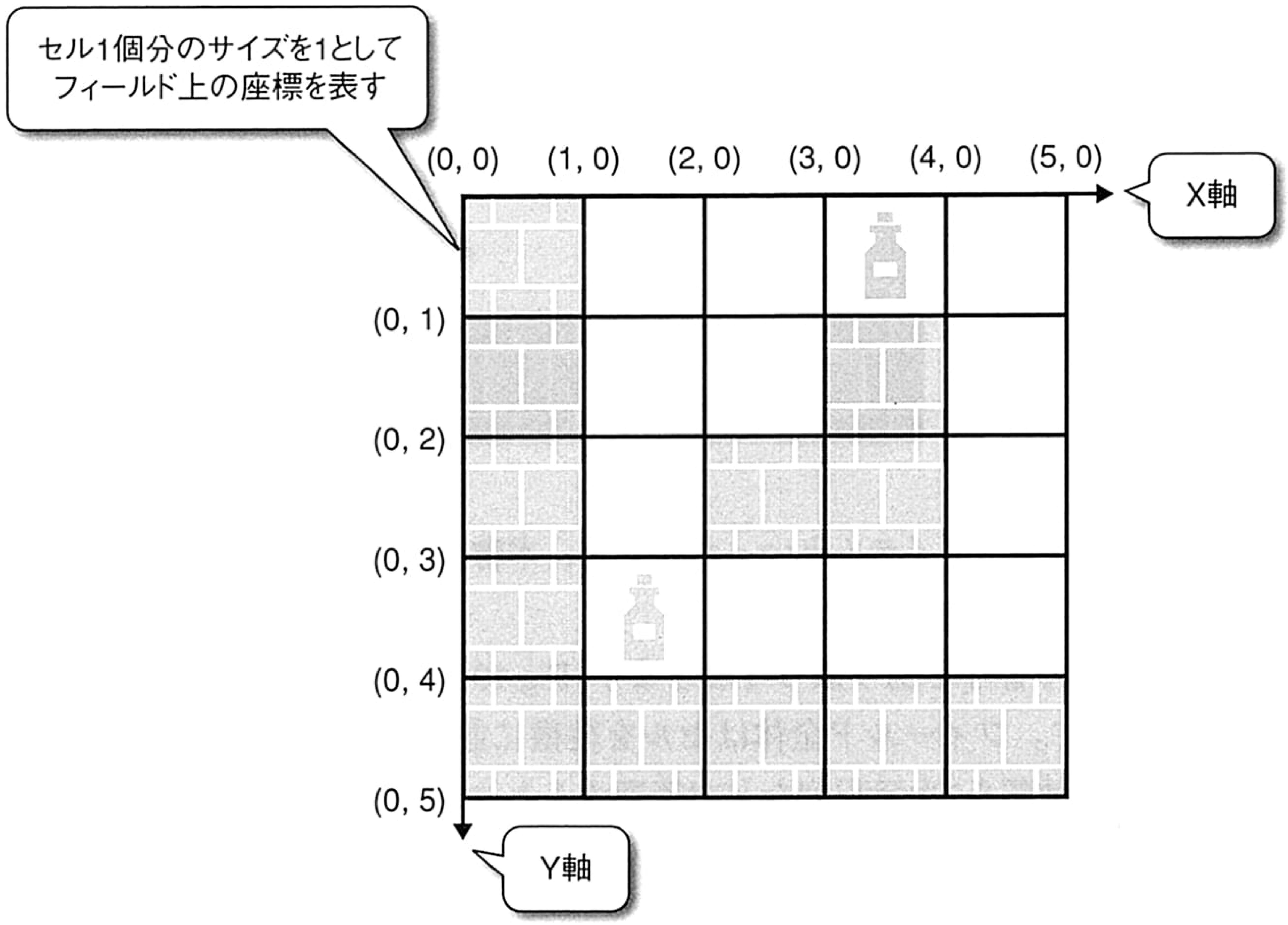
▼ **Fig.10**: セルを並べてフィールドを表現する



セルの取得

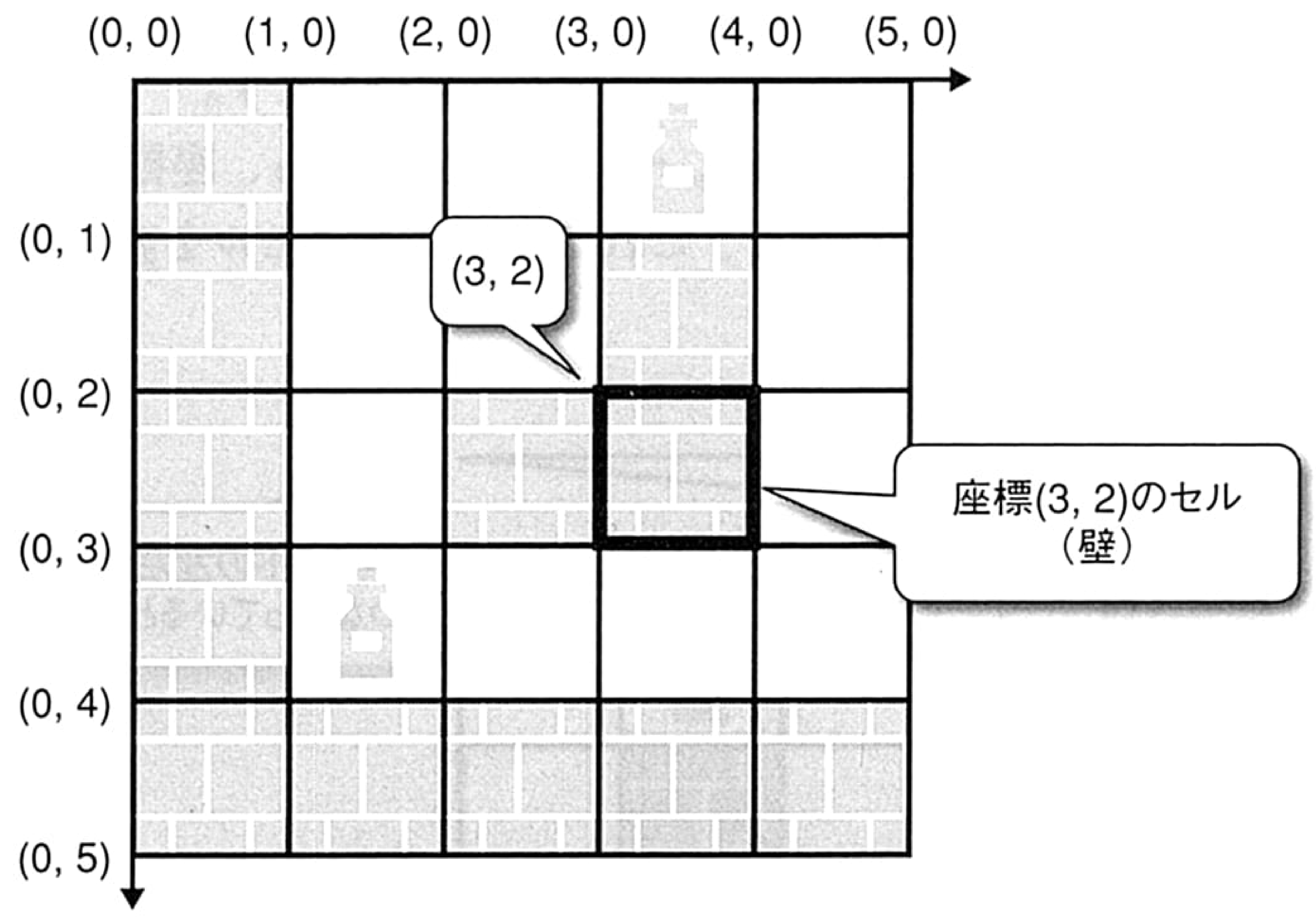
フィールド上の座標は、セル1個の縦横の長さを1として表すことにします(Fig.11)。本書ではセル1個とスプライト1個のサイズを合わせてあるので、フィールドの座標とスプライトの座標は一致します。

▼ Fig.11: フィールドの座標



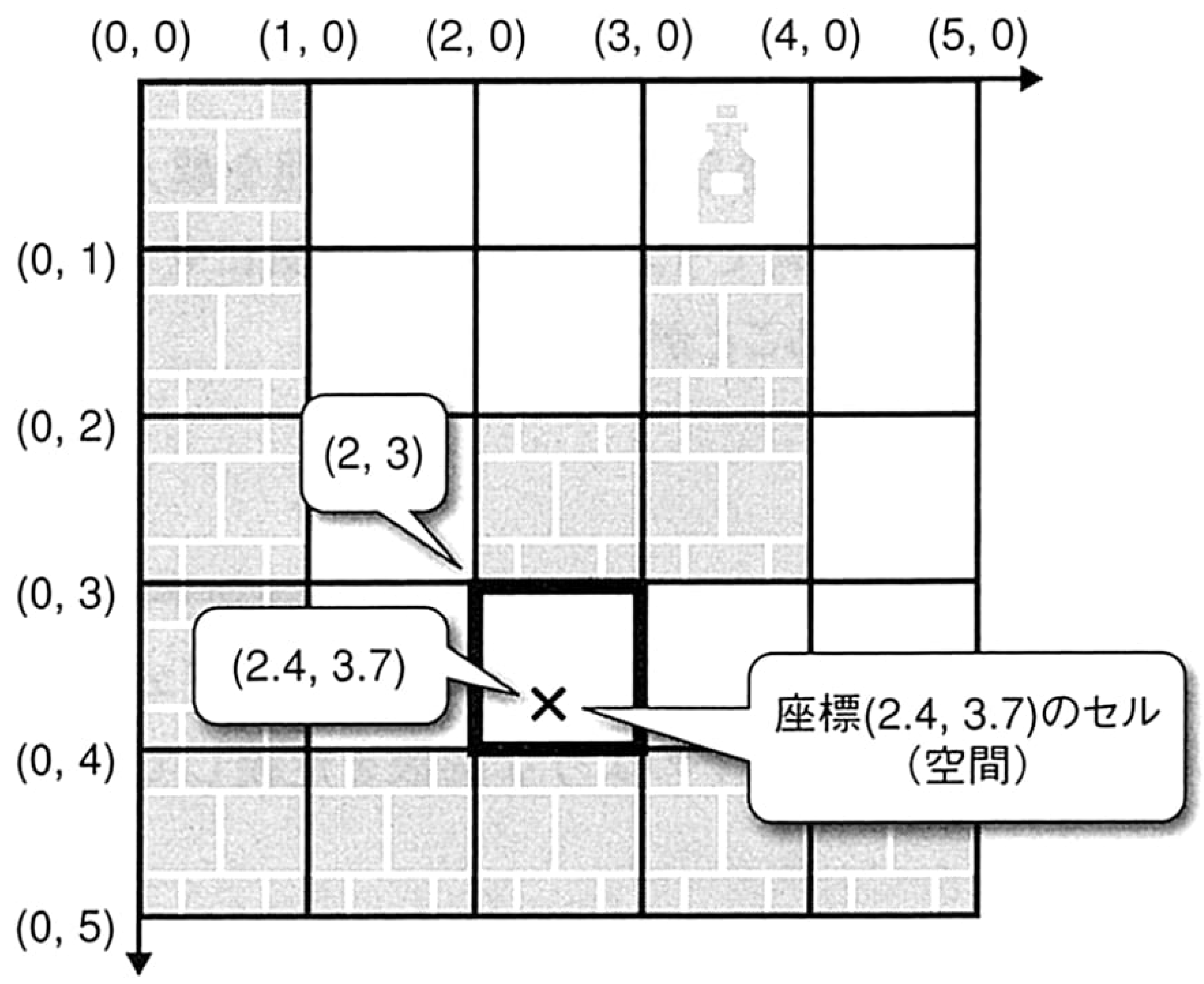
フィールドを表示したり、キャラクターとフィールドの当たり判定処理を行ったりするには、指定した座標のセルを取得する必要があります。例えば座標(3, 2)を指定したら、左端から4番目、上端から3番目のセルを表します(Fig.12)。

▼ **Fig.12**: 指定したセルの取得



座標は整数とは限らず、小数部分を含むことがあります。この場合は、小数部分を切り下げた座標のセルを取得することにします(Fig.13)。例えば座標(2.4, 3.7)を指定したら、座標(2, 3)のセルを表します。プログラムで小数の切り下げを行うには、Math.floor関数(指定した数を超えない最大の整数を返す関数)などを使います。

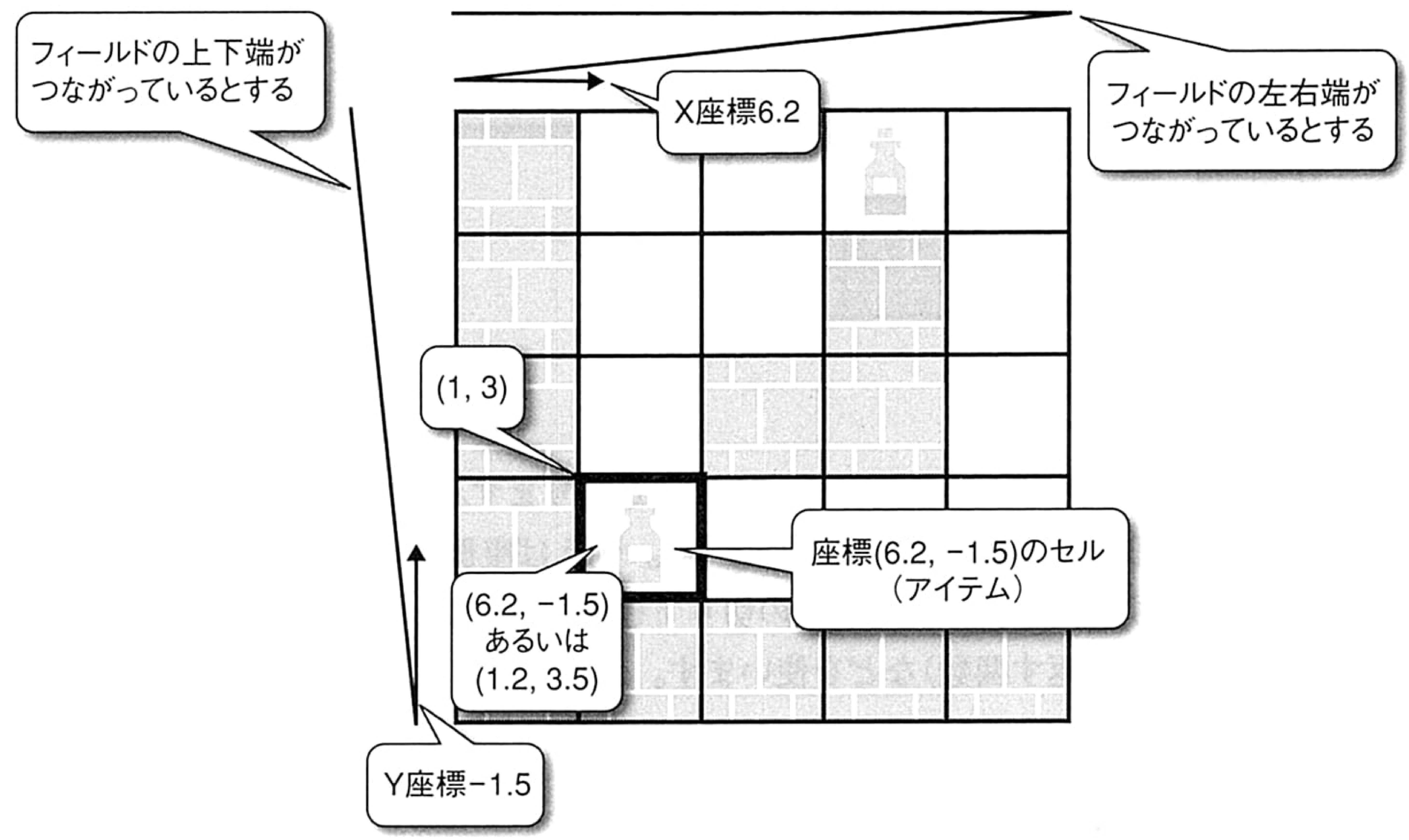
▼ **Fig.13**: 小数部分を含む座標



フィールドの範囲を越える座標を指定する可能性もあります。この場合は、フィールドの両端がつながっており、フィールド上をループできると解釈することにします(Fig.14)。座標をフィールドのサイズ(縦横のセル数)で割った余りを使って、セルを取得します。

例えばフィールドのサイズが5×5のときに、座標(6.2, −1.5)を指定したら、座標(1.2, 3.5)を指定したようにみなします。小数部分を切り下げると、これは座標(1, 3)のセルを表します。

▼ Fig.14: フィールドの範囲を越える座標



このようにフィールドの座標に小数を使えるようにして、さらにフィールドの範囲を越えても構わないようにしておくと、キャラクターを動かすときにとても便利です。セルの配置を気にせずに、キャラクターを細かく動かしたり、広い範囲を移動させたりすることができるからです。

フィールドを管理するプログラム

List1はフィールドを管理するプログラムです。フィールドの機能はFieldオブジェクトにまとめました。ここではセルの初期化や取得などを行う、以下のような関数を定義しています。

- initCell関数　：セルを初期化する
- getCell関数　：指定した座標のセルを取得する
- setCell関数　：指定した座標のセルを設定する
- getXCount関数：フィールドの横方向のセル数を取得する
- getYCount関数：フィールドの縦方向のセル数を取得する

initCell関数では、引数に指定された配列を使って、フィールドのセルを初期化します。引数の配列は、例えば、

```
new Array(
    "10020",
    "10010",
    "10110",
    "12000",
    "11111"
)
```

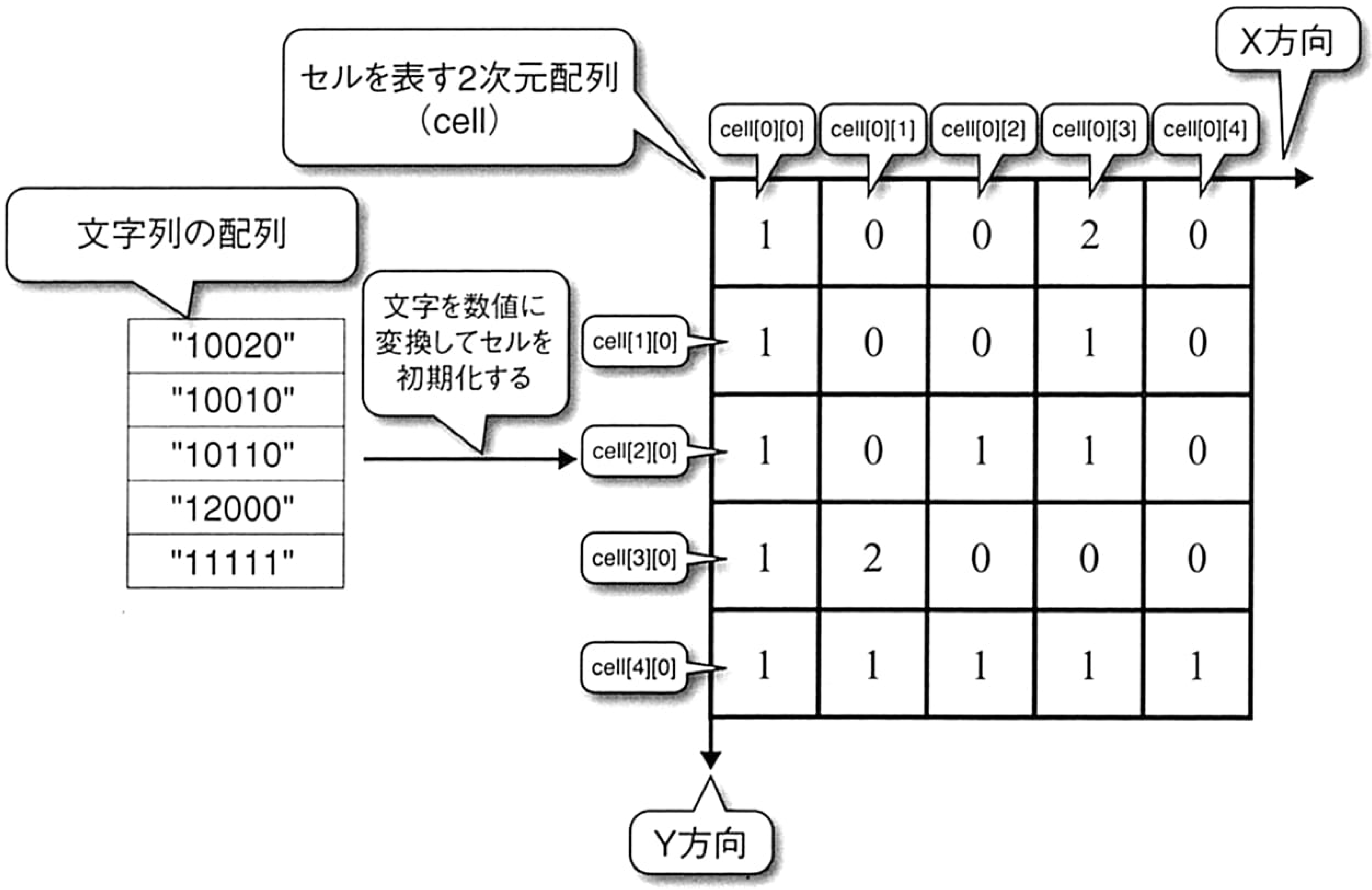
のような文字列の配列です。これは例えば、

```
new Array(
    1, 0, 0, 2, 0,
    1, 0, 0, 1, 0,
    1, 0, 1, 1, 0,
    1, 2, 0, 0, 0,
    1, 1, 1, 1, 1
)
```

のような数値の配列を使ってもよかったのですが、,(カンマ)が多くて書きにくいので、文字列の配列を使うことにしました。

initCell関数は、このような文字列の配列を解釈してセルを初期化します。セルは数値の2次元配列です(Fig.15)。文字列の配列から1個ずつ文字を取り出し、数値に変換してから、セルの配列に格納します。

Fig.15: 文字列の配列を使ってセルを初期化する



文字から数値への変換は、

```
"0123456789abcdef".indexOf(array[y].charAt(x))
```

のような式で行います。ここで、

```
array[y].charAt(x)
```

は、文字列の配列から文字を1個取り出します。例えば文字'0'を取り出したとすると、

```
"0123456789abcdef".indexOf('0')
```

となります。

このindexOf関数は、引数の文字が文字列の中に現れる位置を返します。この場合は、"0123456789abcdef"という文字列の中に、'0'という文字が現れる位置を返すので、結果は0になります。同じ要領で、'0'～'9'の文字を0～9の数値に、'a'～'f'の文字を10～15の数値に変換します。

なお、C/C++/Javaでは文字を'(シングルクォート)で表し、文字列を"(ダブルクォート)で表しますが、JavaScriptではこれらの区別がありません。'と"のどちらでも好きな方を使って、文字や文字列を表すことができます。本書ではC/C++/Javaの流儀にならって、文字は'で、文字列は"で表しています。

ところで、indexOf関数を使う代わりに、

```
array[y].charAt(x)-'0'
```

のような式を使っても、文字から数値への変換ができそうに思えます。確かに変換はできるのですが、この方法は隣接する文字('0'と'1'など)について、文字コードも隣接しているという前提に基づいています。例えばUnicodeでは、'0'は0030(16進数)、'1'は0031なので、確かに文字コードも隣接しています。しかし、使用する文字コードによってはこの前提が成り立たないので、文字コードに依存しない前述のindexOf関数を使った方法が、安全性の点でお勧めです。

次にgetCell関数については、

```
var yc=obj.yCount;
var cy=(Math.floor(y)%yc+yc)%yc;
var xc=obj.xCount;
var cx=(Math.floor(x)%xc+xc)%xc;
```

のような処理で、引数の座標(x, y)に対応するセルの位置を計算します。xcとycはフィールドのサイズ(セル数)で、(cx, cy)はセルの位置です。

まずMath.floor関数を使って、座標の小数部分を切り下げます。例えば座標(x, y)が(8.5, −6.7)のときには、

```
var cy=(8%yc+yc)%yc;
var cx=(-7%xc+xc)%xc;
```

となります。Math.floor関数は引数を超えない最大の整数を返すので、8.5は8になり、−6.7は−7になります。

次に%演算子を使って、座標がフィールドの範囲を越えているときに、フィールド上をループさせます。例えばxcとycが5とすると、

```
var cy=(8%5+5)%5;
var cx=(-7%5+5)%5;
```

となります。さらに計算を進めると、

```
var cy=(3+5)%5;
var cx=(-2+5)%5;
```

となり、結果は、

```
var cy=3;
var cx=3;
```

となります。剰余·加算·剰余という順番で計算するのは、剰余の結果が負の数になることがあるので、これを補正するためです。getCell関数と同様にsetCell関数でも、座標からセルの位置を計算します。

getXCount関数とgetYCount関数は、フィールドのサイズを表すxCountプロパティとyCountプロパティを返します。これらのプロパティはinitCell関数で、初期化に用いた配列のサイズにあわせて設定します。

List1: フィールドの管理(common.js)

```
// フィールド
function Field(div, zIndex) {
  // ...(中略)...
```

```
// セルの初期化
obj.initCell=function(array) {

  // 表示の更新が必要なことを示す
  obj.updated=false;

  // セルの配列を作成する
  obj.cell=new Array();

  // 引数に指定された配列のサイズから、
  // 縦方向のセル数を設定する
  obj.yCount=array.length;

  // 横方向のセル数は仮に整数の最大値としておく
  obj.xCount=Number.MAX_VALUE;

  // 配列の内容からセルの種類を設定する
  for (y=0; y<obj.yCount; y++) {
    obj.cell[y]=new Array();

    // 引数に指定された配列の2次元目の最小サイズを
    // 横方向のセル数にする
    if (obj.xCount>array[y].length) {
      obj.xCount=array[y].length;
    }
    for (x=0; x<obj.xCount; x++) {

      // 配列の内容(0〜9とa〜fの文字)を
      // セルの種類(0〜15の数値)に変換する
      obj.cell[y][x]=
        "0123456789abcdef".indexOf(array[y].charAt(x))
    }
  }
}

// セルの種類を取得する
obj.getCell=function(x, y) {

  // 座標がフィールド上を循環するようにする
  var yc=obj.yCount;
  var cy=(Math.floor(y)%yc+yc)%yc;
  var xc=obj.xCount;
  var cx=(Math.floor(x)%xc+xc)%xc;

  // 指定されたセルの種類を返す
  return obj.cell[cy][cx];
}
```

```
  // セルの種類を設定する
  obj.setCell=function(x, y, value) {

    // 表示の更新が必要なことを示す
    obj.updated=false;

    // 座標がフィールド上を循環するようにする
    var yc=obj.yCount;
    var cy=(Math.floor(y)%yc+yc)%yc;
    var xc=obj.xCount;
    var cx=(Math.floor(x)%xc+xc)%xc;

    // 指定されたセルの種類を設定する
    obj.cell[cy][cx]=value;
  }

  // 横方向のセル数を取得する
  obj.getXCount=function() {
    return obj.xCount;
  }

  // 縦方向のセル数を取得する
  obj.getYCount=function() {
    return obj.yCount;
  }

  // 最初はセル数を1×1とし、セルの種類は0とする
  obj.initCell(new Array("0"));
}
```

フィールドの作成

スプライトと同様に、フィールドはゲーム本体(Gameオブジェクト)で管理することにします。List2はFieldオブジェクトを作成するcreateField関数です。

Gameオブジェクトでは、fieldプロパティにフィールドの一覧を格納します。createField関数を呼び出すと、新しいFieldオブジェクトを作成するとともに、fieldプロパティ(配列)の末尾にオブジェクトを追加します。

List2: フィールドの作成(common.js)

```
// ゲーム本体
function Game() {
  // ...(中略)...
```

```
    // フィールドの作成
    obj.createField=function() {

      // Fieldオブジェクトを作成する
      var f=new Field(obj.divField, obj.zIndex++);

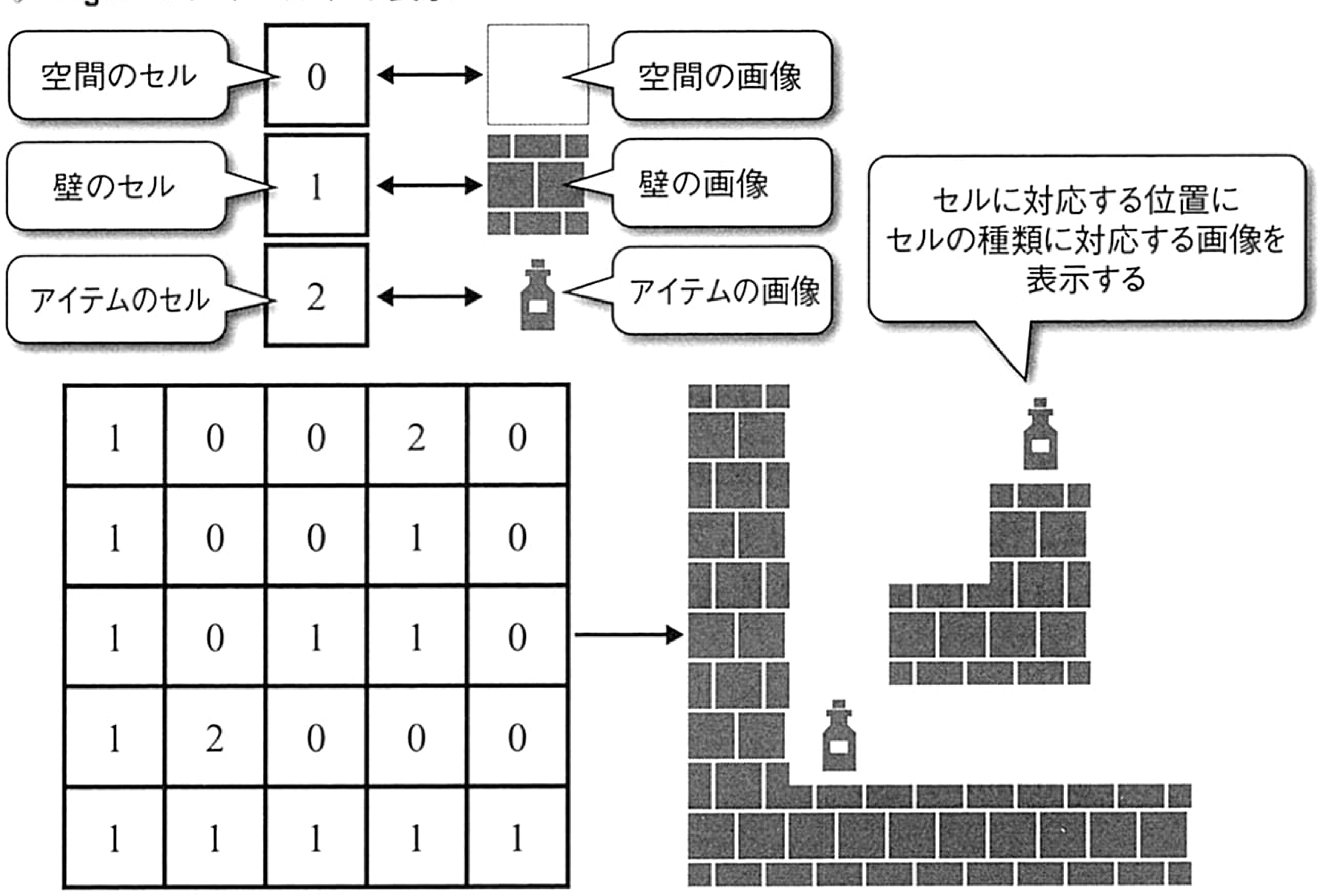
      // fieldプロパティ(配列)の末尾に追加する
      obj.field.push(f);

      // 作成したFieldオブジェクトを返す
      return f;
    }
  }
```

フィールドの表示

フィールドの管理方法が決まったところで、次はフィールドを画面に表示する方法について考えましょう。基本的には、セルに対応する位置に、セルの種類に対応する画像を表示するだけです(Fig.16)。

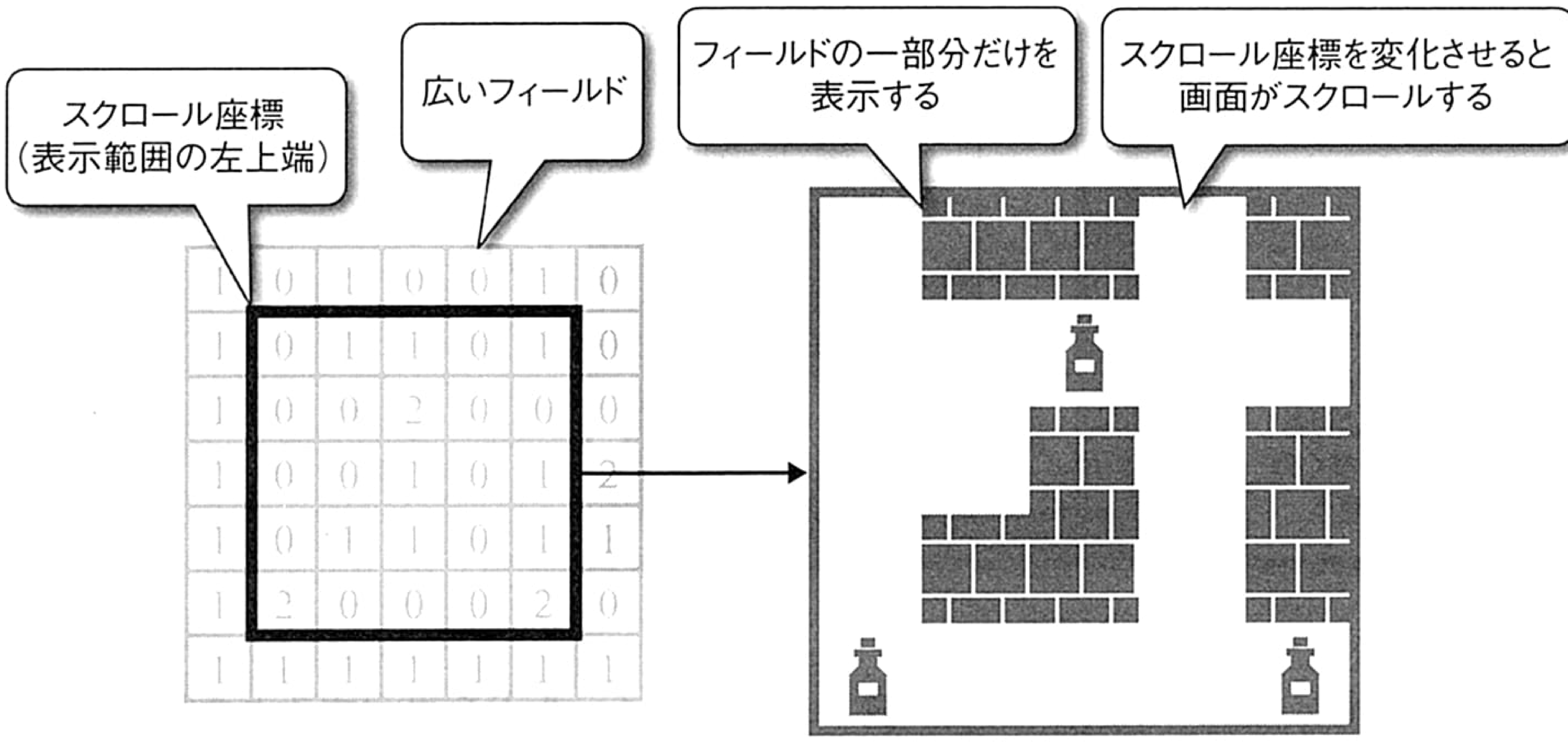
▼ **Fig.16**: フィールドの表示



問題はスクロールです。フィールドが広くて画面に収まらないときには、フィールドの一部分だけを表示します(Fig.17)。

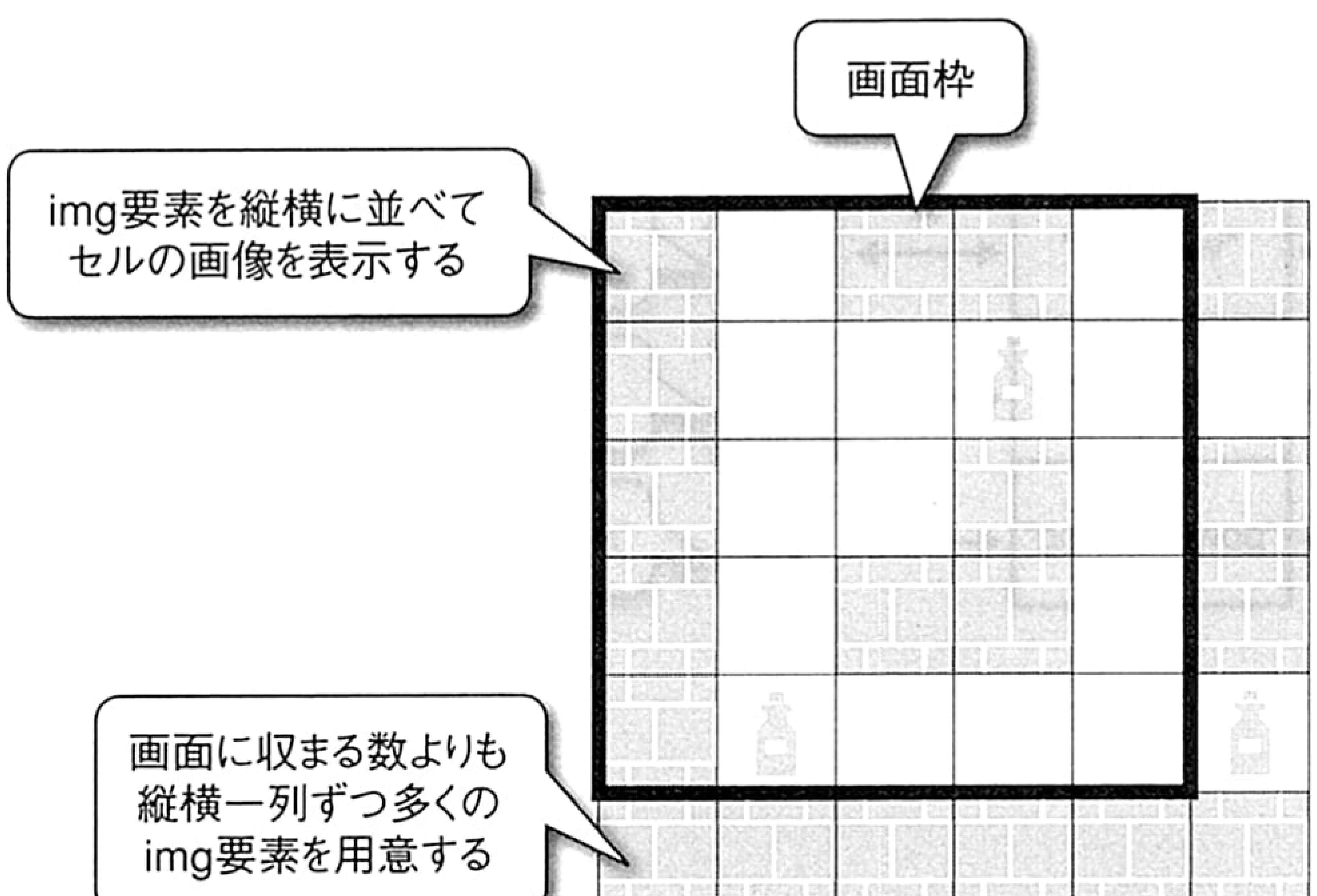
フィールドを表示する範囲は、左上端の座標で表すことにしましょう。この座標をスクロール座標と呼ぶことにします。スクロール座標を変化させると、フィールドをスクロールさせながら表示することができます。

▼ **Fig.17**: スクロールの実現



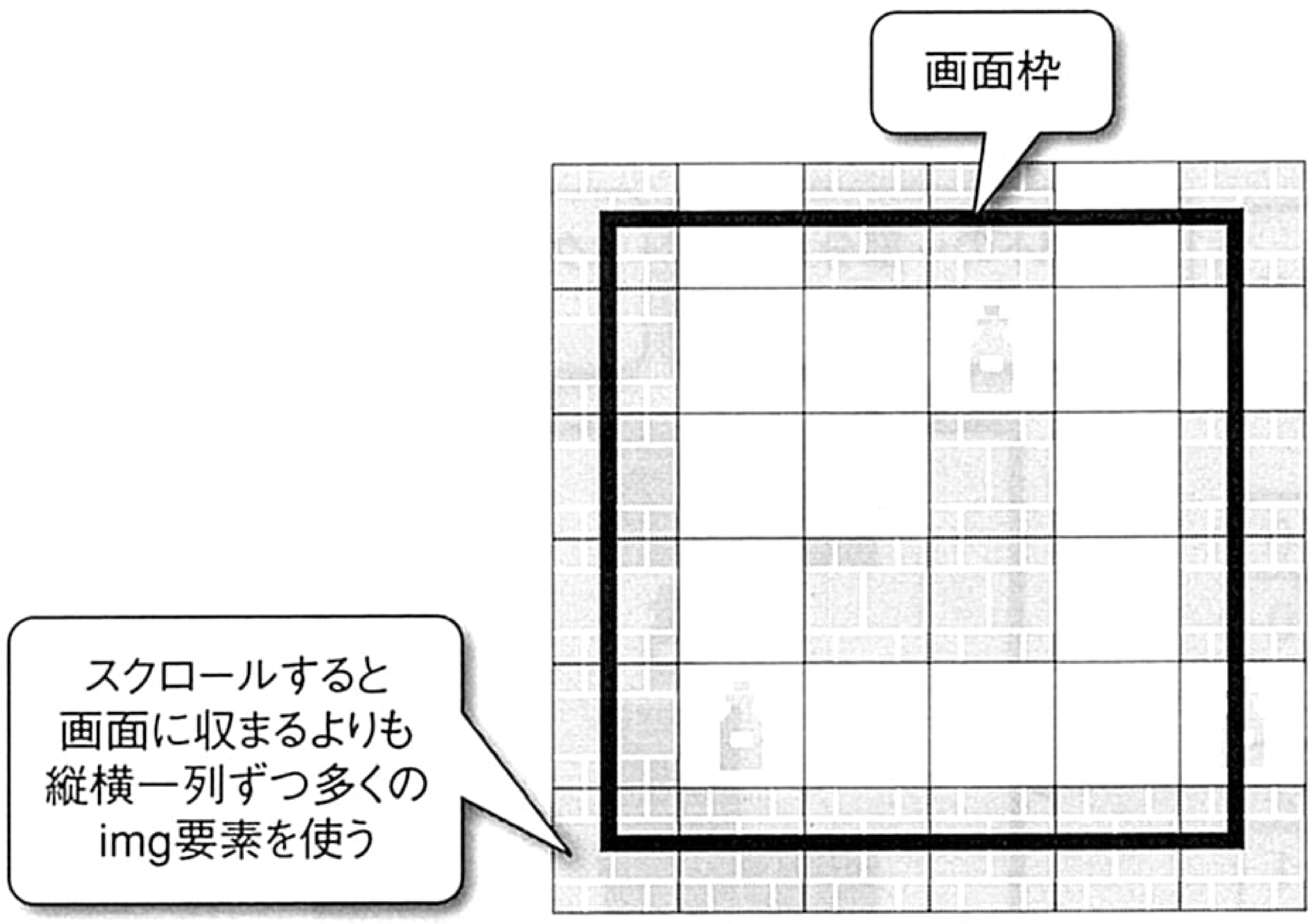
画像を表示するには、スプライト(p.60)と同様にHTMLのimg要素を使います。img要素を縦横に並べて、セルの種類に応じた画像を表示することにより、フィールドを表示します(Fig.18)。

▼ **Fig.18**: img要素を使った画像の表示



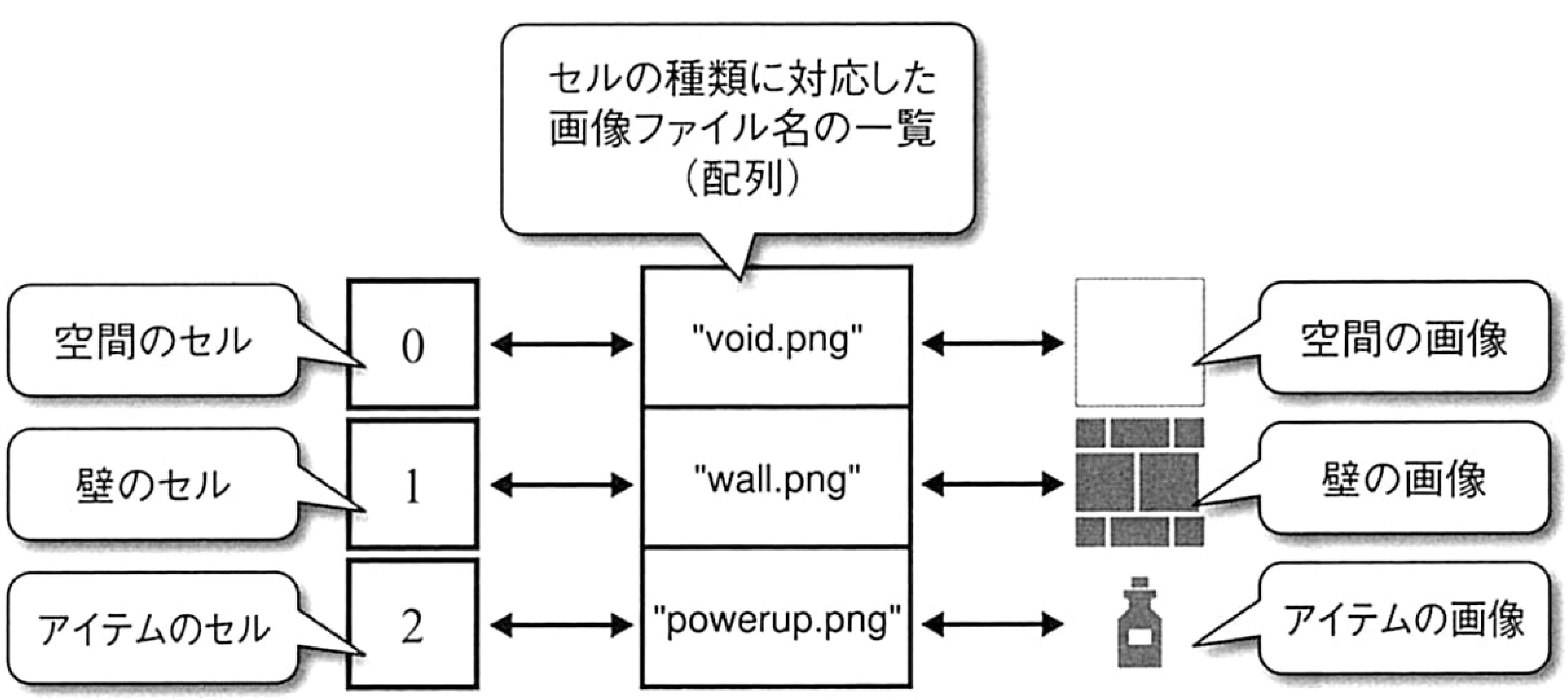
img要素はフィールド全体のセル数と同じだけ用意する必要はなく、画面に収まる数だけ用意すれば十分です。スクロールを行うとimg要素の一部が画面枠にかかることがあるので、画面に収まる数よりも縦横一列ずつ多くのimg要素を用意します(Fig.19)。

Fig.19: 縦横一列ずつ多くのimg要素を用意する



前述のように、セルの種類は数値で表します。フィールドを表示するときには、例えば1ならば壁の画像、2ならばアイテムの画像といった要領で、セルの種類に対応した画像を表示します。これは画像ファイル名の一覧を配列などで用意しておけばよいでしょう。

Fig.20: セルの種類と画像の対応



フィールドを表示するプログラム

List3はフィールドを表示するプログラムです。フィールドを表示する処理はFieldオブジェクトに含めました。

フィールドの表示に使うimg要素は、

```
obj.img=new Array();
```

のようなimgプロパティに格納します。imgプロパティは2次元配列です。img要素は、

```
obj.img[y][x]=document.createElement("img");
```

のように作成し、

```
div.appendChild(obj.img[y][x]);
```

のようにdiv要素へ登録することによって、画面に表示します。フィールド表示用のdiv要素は、Gameオブジェクトで作成します(p.51)。

img要素に画像を表示するには、srcプロパティを設定します。最初は、

```
obj.img[y][x].src="void.png";
```

のように、空の画像(void.png)を表示します。また、

```
style=obj.img[y][x].style;
style.position="absolute";
style.zIndex=zIndex;
```

のように、座標の指定方法(絶対座標で指定)と、重ね合わせの順番も設定します。これはスプライトのimg要素に設定した内容と同じです(p.61)。

フィールドを表示するのはupdate関数です。ここではまず、

```
if (
    obj.updated &&
    viewX==obj.viewX && viewY==obj.viewY
) return;
```

のように、表示を更新する必要があるかどうかを判定し、必要がなければ更新を省略します。フィールドの表示を更新するのは重い処理なので、本当に必要なときだけ更新します。

updatedは更新の必要があるかどうかを表すプロパティです。セルを設定したときや、セルに対応する画像を入れ替えたときなどに、trueを設定します。

(viewX, viewY)はスクロール座標です。スクロール座標が変化したときには、表示を更新する必要があります。

表示を更新する場合には、

```
obj.img[y][x].src=
    obj.cellImage[obj.getCell(viewX+x, viewY+y)];
```

のように、セルに対応する画像をimg要素に設定します。cellImageプロパティは、セルに対応する画像ファイル名の一覧です。このプロパティはinitCellImage関数を使って初期化します。

次にimg要素の座標を、

```
style=obj.img[y][x].style;
style.left=Math.floor(
    (Math.floor(viewX)-viewX+x)*CELL_SIZE)+"px";
style.top=Math.floor(
    (Math.floor(viewY)-viewY+y)*CELL_SIZE)+"px";
```

のように設定します。ここで、

```
Math.floor(viewX)-viewX
Math.floor(viewY)-viewY
```

というのは、(viewX, viewY)の小数部分の符号を反転したものを表します。例えば(viewX, viewY)が(0.3, 1.2)のときには、

```
Math.floor(0.3)-0.3
Math.floor(1.2)-1.2
```

となります。Math.floor関数は引数を越えない最大の整数を返すので、

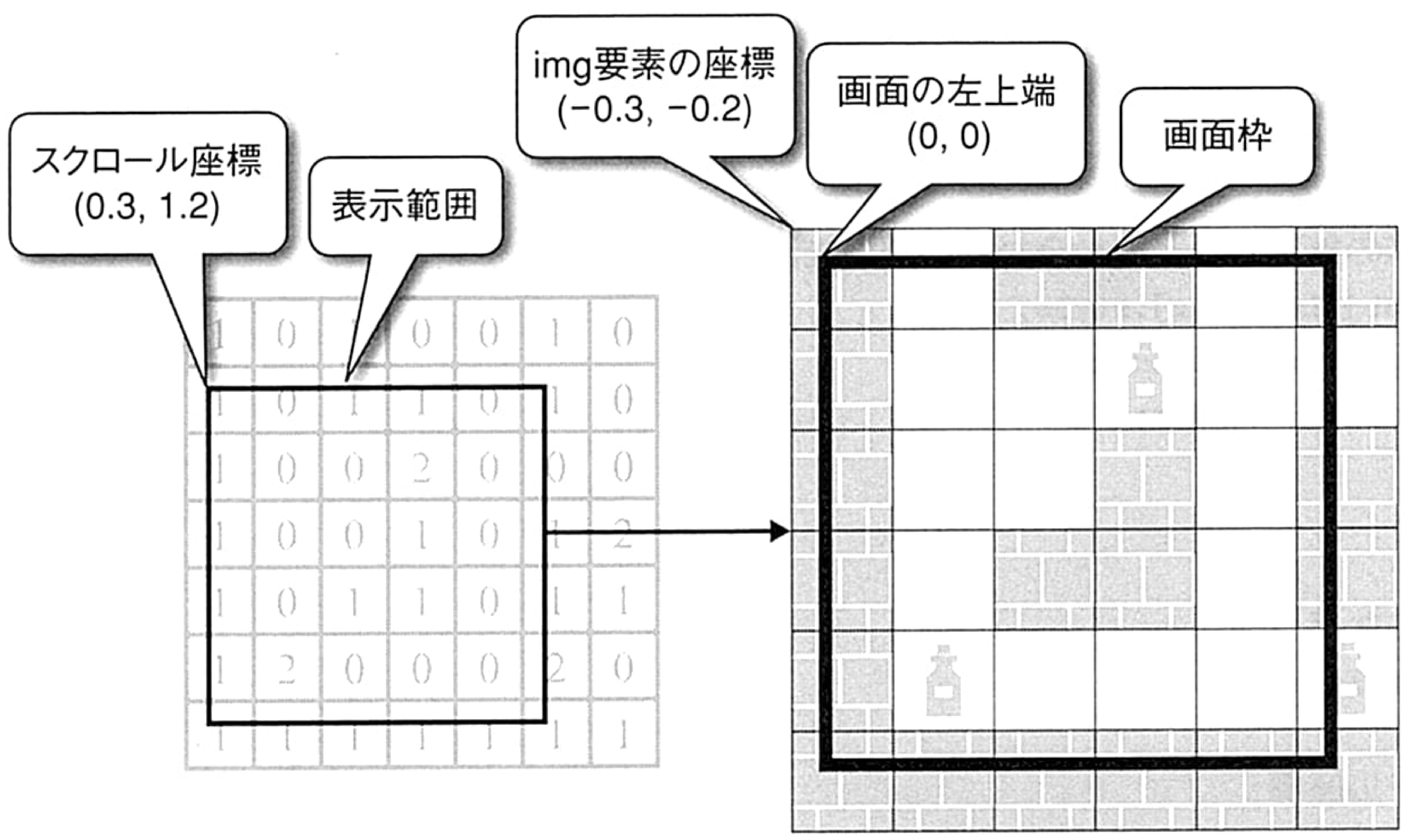
```
0-0.3
1-1.2
```

となり、結果は、

```
-0.3
-0.2
```

となります。これは(0.3, 1.2)の小数部分(0.3, 0.2)の符号を反転したものです。この値を使ってimg要素の座標をずらすと、フィールドを細かくスクロールさせることができます(Fig.21)。

▼ **Fig.21**: img要素の座標をずらす



なおList3では、

```
obj.onUpdate=function() {}
```

のようなonUpdate関数を定義しています。これはフィールドの表示を更新する際に呼び出す関数です。スプライトの場合(p.67)と同様に、onUpdate関数の内容を定義すれば、ゲームごとに独自の処理を行うことができます。

▼ **List3**: フィールドの表示(common.js)

```
// フィールド
function Field(div, zIndex) {
  var obj=this;
  var style;
  var x, y;

  // フィールド表示用のimg要素を作成する
  obj.img=new Array();
  for (y=0; y<CELL_YCOUNT+1; y++) {
    obj.img[y]=new Array();
    for (x=0; x<CELL_XCOUNT+1; x++) {

      // img要素を作成し、2次元配列に格納する
      obj.img[y][x]=document.createElement("img");

      // img要素をdiv要素に登録する
      div.appendChild(obj.img[y][x]);

      // 最初は空の画像を表示しておく
      obj.img[y][x].src="void.png";

      // 座標の指定方法と重ね合わせの順番を設定する
      style=obj.img[y][x].style;
      style.position="absolute";
      style.zIndex=zIndex;
    }
  }

  // 更新済みかどうか
  obj.updated=false;

  // スクロール座標
  obj.viewX=0;
  obj.viewY=0;

  // フィールドの表示更新時に呼び出す処理
  obj.onUpdate=function() {}

  // 表示の更新
  obj.update=function(viewX, viewY) {

    // 更新の必要がなく、スクロール座標も前回と同じならば、
    // 表示の更新を省略する
    if (
      obj.updated &&
      viewX==obj.viewX && viewY==obj.viewY
```

```
    ) return;

    // 更新済みにする
    obj.updated=true;

    // スクロール座標を保存する
    obj.viewX=viewX;
    obj.viewY=viewY;

    // 全てのimg要素について表示を更新する
    var style;
    var x, y;
    for (y=0; y<obj.img.length; y++) {
      for (x=0; x<obj.img[y].length; x++) {

        // 画像の設定
        obj.img[y][x].src=
          obj.cellImage[obj.getCell(viewX+x, viewY+y)];

        // 座標の設定
        style=obj.img[y][x].style;
        style.left=Math.floor(
          (Math.floor(viewX)-viewX+x)*CELL_SIZE)+"px";
        style.top=Math.floor(
          (Math.floor(viewY)-viewY+y)*CELL_SIZE)+"px";
      }
    }
  }

  // セルの種類に対応する画像の一覧を配列で設定する
  obj.initCellImage=function(array) {
    obj.updated=false;
    obj.cellImage=array;
  }

  // 最初は空の画像を設定する
  obj.initCellImage(new Array("void.png"));

  // ...(中略)...
}
```

フィールドを一定時間ごとに更新する

スプライトと同様に、フィールドも一定時間ごとに表示の更新処理（List3のupdate関数）を呼び出す必要があります。これはList4のように、Gameオブジェクトのupdate関数で行います。この関数は、

```
setInterval(obj.update, UPDATE_INTERVAL);
```

のように、一定時間（本書では50ミリ秒）ごとに実行されます（p.67）。

update関数では、まず、

```
for (i=0; i<obj.field.length; i++) {
    obj.field[i].onUpdate();
}
```

のように、フィールドのonUpdate関数を呼び出します。onUpdate関数には、ゲームごとに独自の更新処理を記述することができます（p.195）。

次に、

```
for (i=0; i<obj.field.length; i++) {
    obj.field[i].update(obj.viewX, obj.viewY);
}
```

のようにフィールドのupdate関数を呼び出して、表示を更新します。(obj.viewX, obj.viewY)はスクロール座標です。

▼ **List4**: フィールドを一定時間ごとに更新する（common.js）

```
// ゲーム本体
function Game() {
  // ...(中略)...

  // 更新時に行いたい独自の処理を記述する関数
  obj.onUpdate=function() {}

  // 表示の更新
  obj.update=function() {

    // ゲーム本体の更新処理を呼び出す
    obj.onUpdate();

    // フィールドの更新処理を呼び出す
```

```
    var i;
    for (i=0; i<obj.field.length; i++) {
      obj.field[i].onUpdate();
    }

    // スプライトの更新処理を呼び出す
    for (i=0; i<obj.sprite.length; i++) {
      obj.sprite[i].onUpdate();
    }

    // フィールドの表示を更新する
    for (i=0; i<obj.field.length; i++) {
      obj.field[i].update(obj.viewX, obj.viewY);
    }

    // スプライトの表示を更新する
    for (i=0; i<obj.sprite.length; i++) {
      obj.sprite[i].update(obj.viewX, obj.viewY);
    }
  }

  // 一定時間ごとに表示の更新(update関数)を実行する
  setInterval(obj.update, UPDATE_INTERVAL);
}
```

スクロール座標の管理

スクロール座標はGameオブジェクトで管理することにします(List5)。スクロール座標はフィールドの表示に使うだけではなく、スプライトの表示(p.68)やマウスの入力(p.117)でも利用するため、ゲーム本体を表すGameオブジェクトで管理するのが適切だと考えました。

スクロール座標を管理するために、List5では次のような関数を用意しました。

- setViewXY関数　　　　　　　　:スクロール座標を設定する
- getViewX関数, getViewY関数 :スクロール座標を取得する

List5: スクロール座標の管理(common.js)

```
// ゲーム本体
function Game() {
  // ...(中略)...

  // スクロール座標の設定
```

```
  obj.setViewXY=function(x, y) {
    obj.viewX=x;
    obj.viewY=y;
  }

  // スクロール座標の取得(X座標)
  obj.getViewX=function() {
    return obj.viewX;
  }

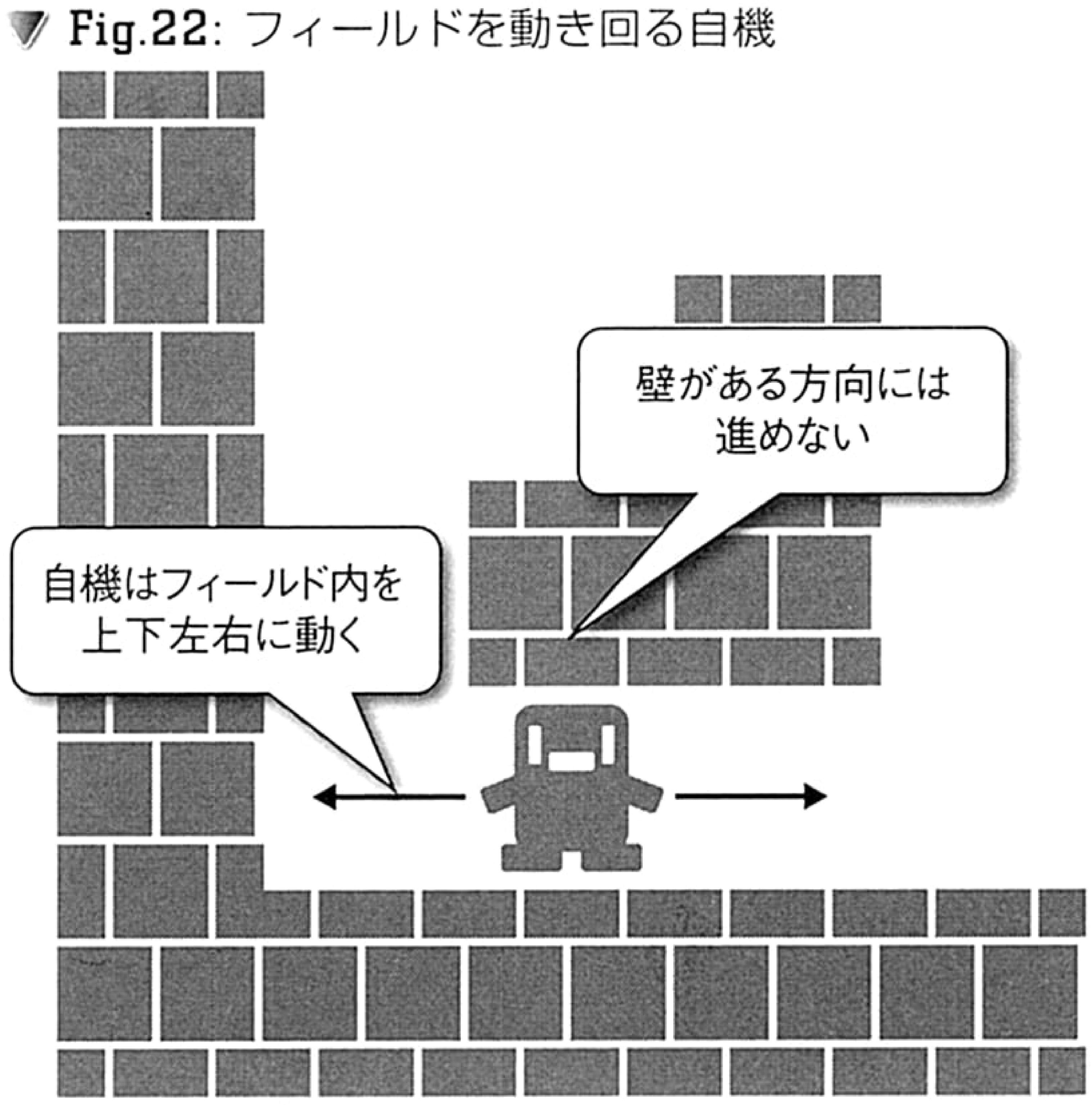
  // スクロール座標の取得(Y座標)
  obj.getViewY=function() {
    return obj.viewY;
  }

  // 最初はスクロール座標を(0, 0)に設定する
  obj.setViewXY(0, 0);
}
```

フィールドを動き回る自機

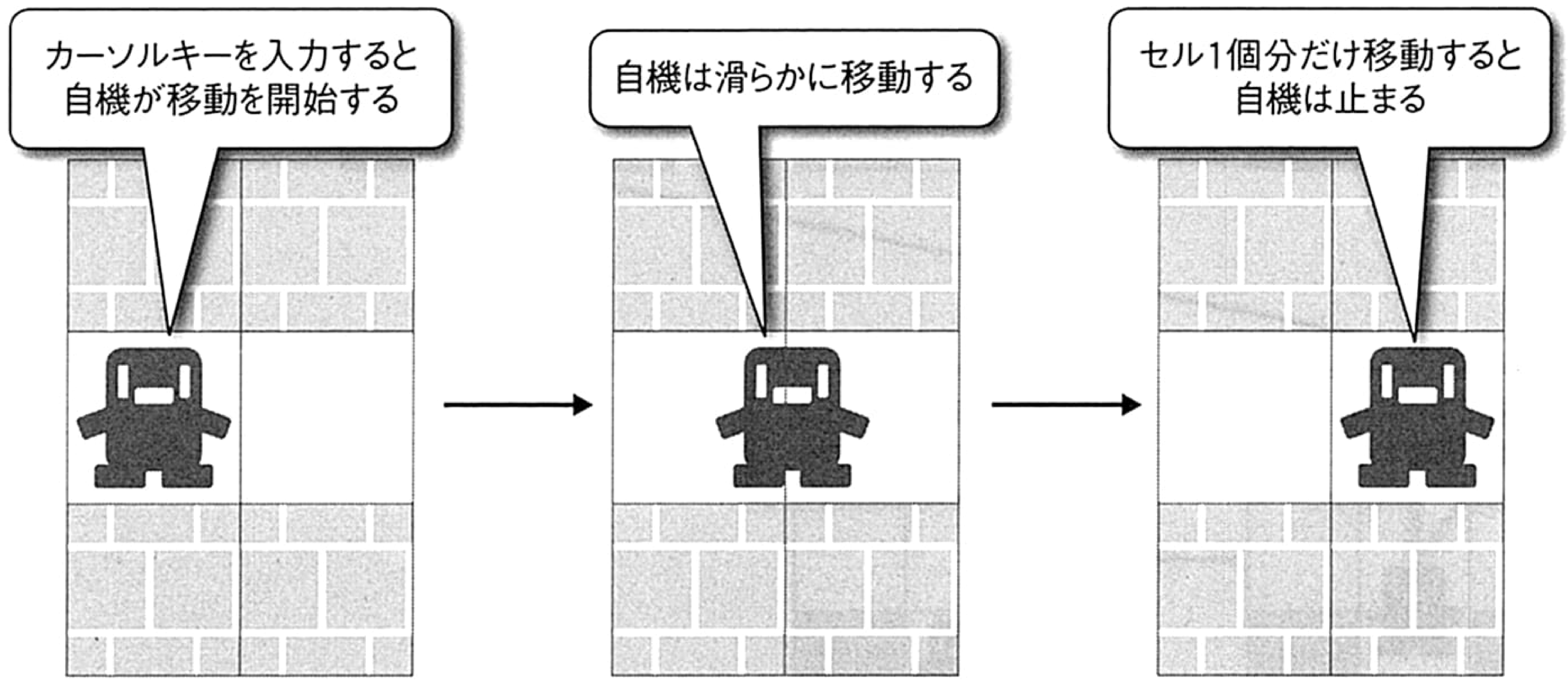
フィールドに関する処理が一通りできたところで、今度は自機を動かしてみましょう。本章のサンプルでは、自機はフィールド内を上下左右に動き回ります(Fig.22)。ただし、自機が動けるのは空間(通路)の部分だけで、壁を乗り越えることはできません。

▼ **Fig.22**: フィールドを動き回る自機



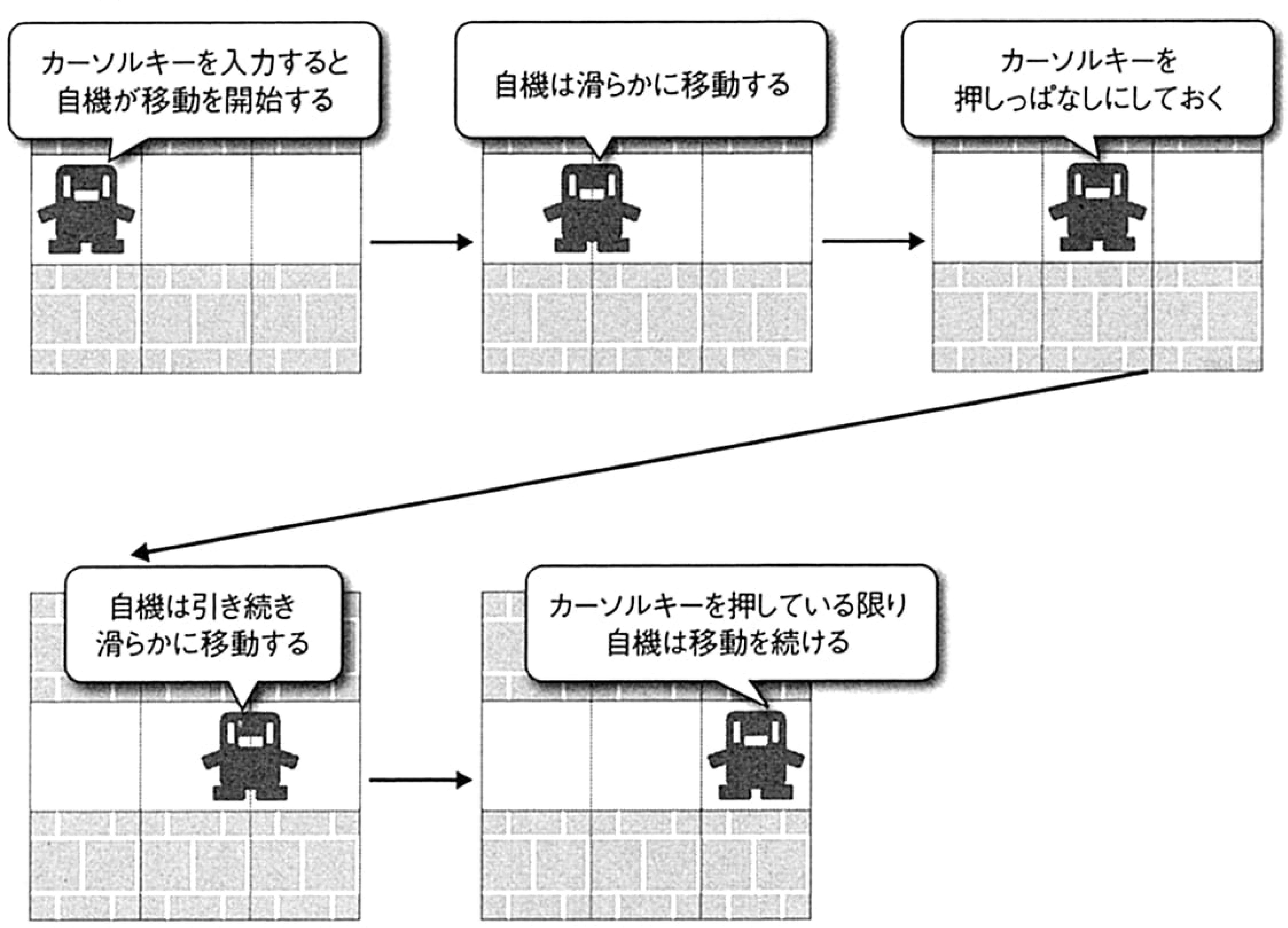
カーソルキーを入力すると、自機はセル1個分だけその方向に移動します(Fig.23)。画面上では自機が滑らかに動き、セル1個分だけ移動すると止まります。

▼ **Fig.23**: セル単位で移動する

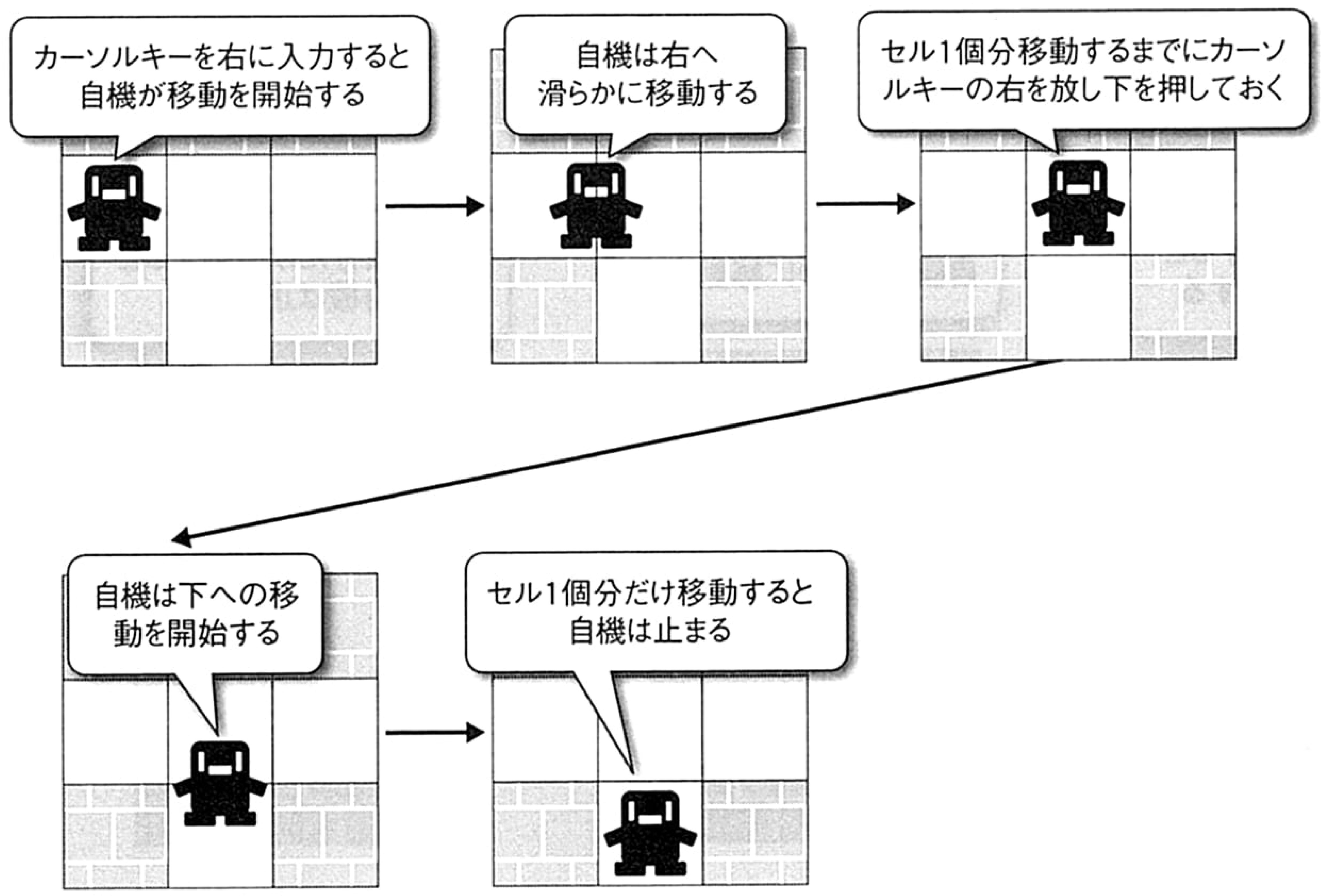


カーソルキーを押しっぱなしにすると、その方向に連続して滑らかに動きます(Fig.24)。また、カーソルキーの別の方向を押しておくと、セル1個分の移動が終わったところで、新しい方向に進みます(Fig.25)。

▼ **Fig.24**: 同じ方向に連続して動く



▼ Fig.25: 新しい方向に進む

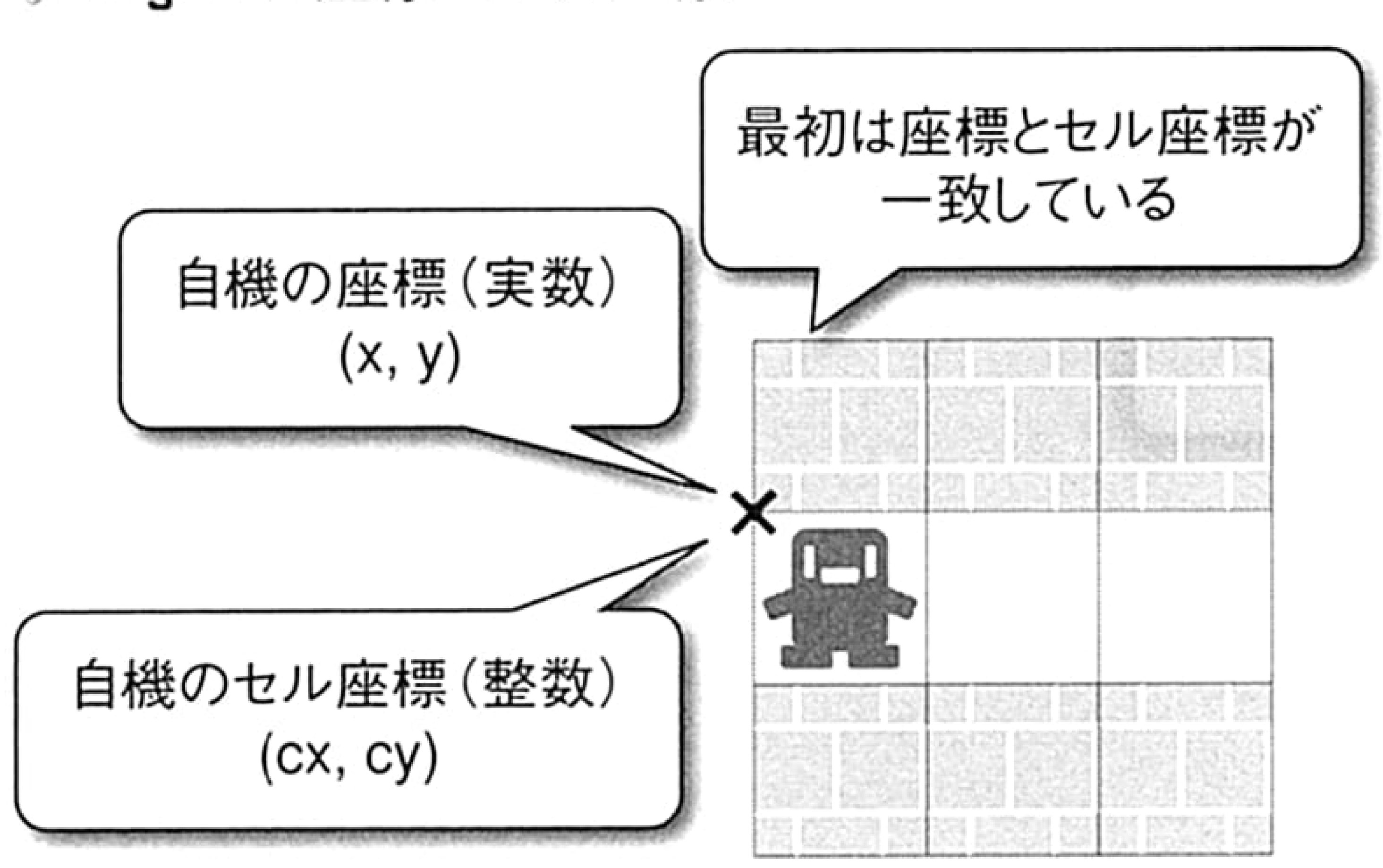


自機をセル単位で動かす

このように自機をセル単位で動かすと、自機が壁に当たったかどうかを簡単に判定することができます。自機を動かす方法について、もう少し具体的に考えてみましょう。

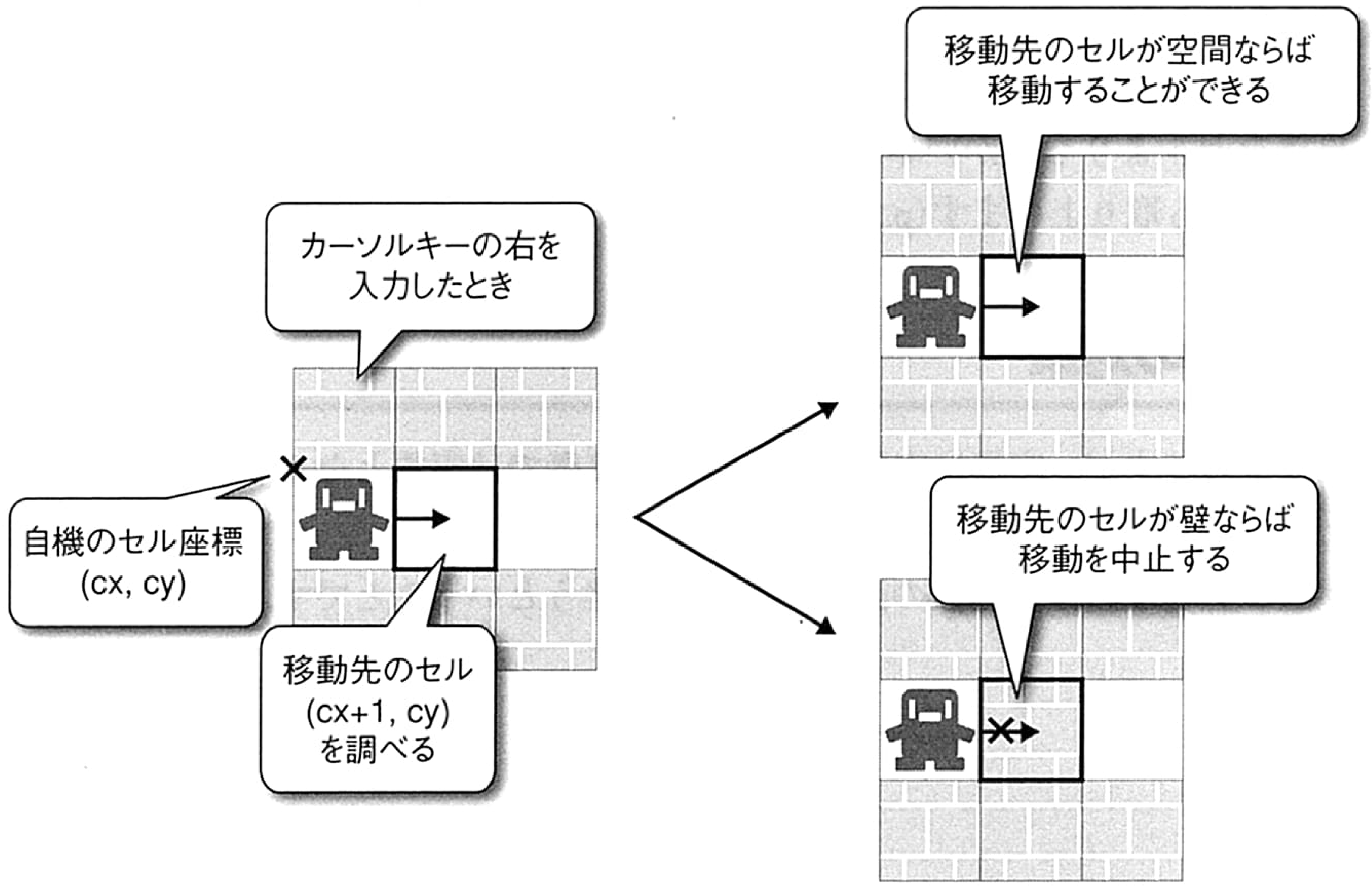
自機の座標を(x, y)とします。また、セル単位の座標をセル座標と呼ぶことにして、自機のセル座標を(cx, cy)とします。座標(x, y)は実数ですが、セル座標(cx, cy)は整数です。最初は座標とセル座標が一致しています(Fig.26)。

▼ Fig.26: 座標とセル座標



カーソルキーを入力したら、移動先のセルが壁かどうかを調べます(Fig.27)。例えばカーソルキーの右を入力したら、座標(cx+1, cy)のセルを調べて、壁の場合には移動を中止します。

▼ Fig.27: 移動先のセルを調べる



移動先が壁でなければ、セル座標(cx, cy)を(cx+1, cy)に更新します(Fig.28)。座標(x, y)については、一気に(x+1, y)にするのではなく、例えば、

(x+0.1, y)

(x+0.2, y)

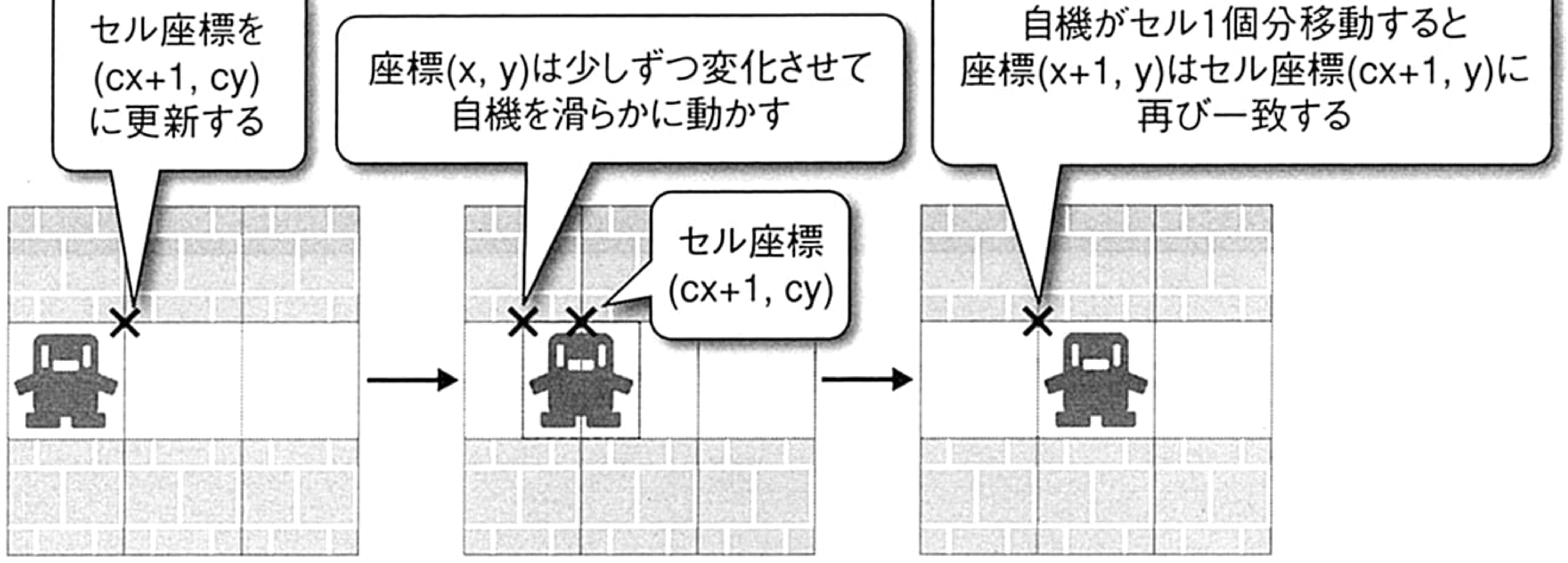
(x+0.3, y)

…

のように、少しずつ変化させます。これは自機を画面上で滑らかに動かすためです。

自機がセル1個分だけ移動すると、座標は(x+1, y)となり、セル座標(cx+1, cy)に再び一致します。あとは今までと同様に、カーソルキーの入力に応じて移動方向を決め、移動先のセルが壁かどうかを調べて、座標とセル座標を更新するという手順を繰り返せば、キャラクターをフィールド内で自由に動かすことができます。

▼ Fig.28: 座標とセル座標の更新



このように自機をセル単位で動かすと、プログラムが簡単になるだけではなく、自機が角を曲がりやすくなるなど、プレイヤーにとっても利点があります。特に荷物を押して運ぶパズルゲームなどでは、自機がセル単位で動いた方が操作しやすくなります(p.259)。

一方、自機をセルよりも細かい単位で動かした方がよいゲームもあります。第9章ではこういったゲームの例も取り上げます(p.273、p.287)。

座標の誤差への対策

セル座標は整数ですが、座標は実数です。自機の移動では座標とセル座標を比べて、両者が一致したときにプレイヤーの入力を受け付けています。

ここで注意が必要なのは、実数の計算には誤差が伴うということです。例えば、

(x+0.1, y)
(x+0.2, y)
(x+0.3, y)
...

のように座標を0.1ずつ変化させると、いずれは、

(x+1, y)

のような座標になると期待されますが、実際には、

(x+1.00000001, y)

や、

(x+0.99999998, y)

といった、微小な誤差のある値になることが一般的です。こういった値と整数を==演算子で比較すると、ほとんど同じ値にもかかわらず、誤差のために一致しないことがあります。実数を扱うときにはこういった誤差について考慮しないと、例えばときどき自機が操作できなくなるといった、不可解なバグが発生します。

誤差の問題を回避するには、

(x+1.00000001, y)
(x+0.99999998, y)

のような値を、

(x+1, y)

のような値に補正する方法があります。この補正は、

```
Math.floor(x*grid+0.5)/grid
```

のような式で行えます。gridは補正の細かさを表す値です。例えば、

(x+0.1, y)
(x+0.2, y)
(x+0.3, y)
...

のように、0.1単位(1/10単位)の値に補正するときには、gridを10にして、

```
Math.floor(x*10+0.5)/10
```

という計算を行います。

例えばxが1.00000001のときには、

```
Math.floor(1.00000001*10+0.5)/10
```

となり、さらに計算すると、

```
Math.floor(10.0000001+0.5)/10
Math.floor(10.5000001)/10
10/10
1
```

のように、結果は1になります。

同様に、例えばxが0.99999998のときには、

```
Math.floor(0.99999998*10+0.5)/10
Math.floor(9.9999998+0.5)/10
Math.floor(10.4999998)/10
10/10
1
```

のように、やはり結果は1になります。このように値を補正しておけば、整数と実数、あるいは実数と実数を==演算子で比較したときに、期待通りの結果を得ることができます。厳密には、整数以外の値については誤差が残るのですが、比較する値を両方とも補正しておけば大丈夫です。

こういった補正をどこで行うのかには色々な考え方がありますが、本書のサンプルではSpriteオブジェクトで行うことにしました(List6)。補正の細かさを表す値(グリッド)という概念を導入して、

・setGrid　　：グリッドを設定する
・getGrid　　：グリッドを取得する

という関数を追加します。そして、スプライトの座標を設定するsetXY関数に対して、

```
obj.x=Math.floor(x*obj.grid+0.5)/obj.grid;
obj.y=Math.floor(y*obj.grid+0.5)/obj.grid;
```

のように、グリッドを使って座標を補正する処理を追加しました。グリッドが0のときには補正を行いません。

▼ **List6**: 座標の補正(common.js)

```
// スプライト
function Sprite(div, zIndex) {
  // ...(中略)...

  // グリッドの設定
  obj.setGrid=function(g) {
    obj.grid=g;
  }

  // グリッドの取得
  obj.getGrid=function() {
    return obj.grid;
  }

  // 最初はグリッドを0にする(補正を行わない)
  obj.setGrid(0);

  // 座標の設定
  obj.setXY=function(x, y) {

    // 表示の更新が必要なことを示す
    obj.updated=false;

    // グリッドが0以外のとき
    if (obj.grid!=0) {

      // 座標を補正してから保存する
      obj.x=Math.floor(x*obj.grid+0.5)/obj.grid;
      obj.y=Math.floor(y*obj.grid+0.5)/obj.grid;
    } else

    // グリッドが0のとき
```

```
      {
        // 補正を行わない
        obj.x=x;
        obj.y=y;
      }
    }
}
```

フィールドを動き回る自機のプログラム

自機をセル単位で動かす方法と、座標を補正する方法を利用して、フィールドを動き回る自機のプログラムを書きました(List7)。自機の処理はPlayerオブジェクトにまとめています。

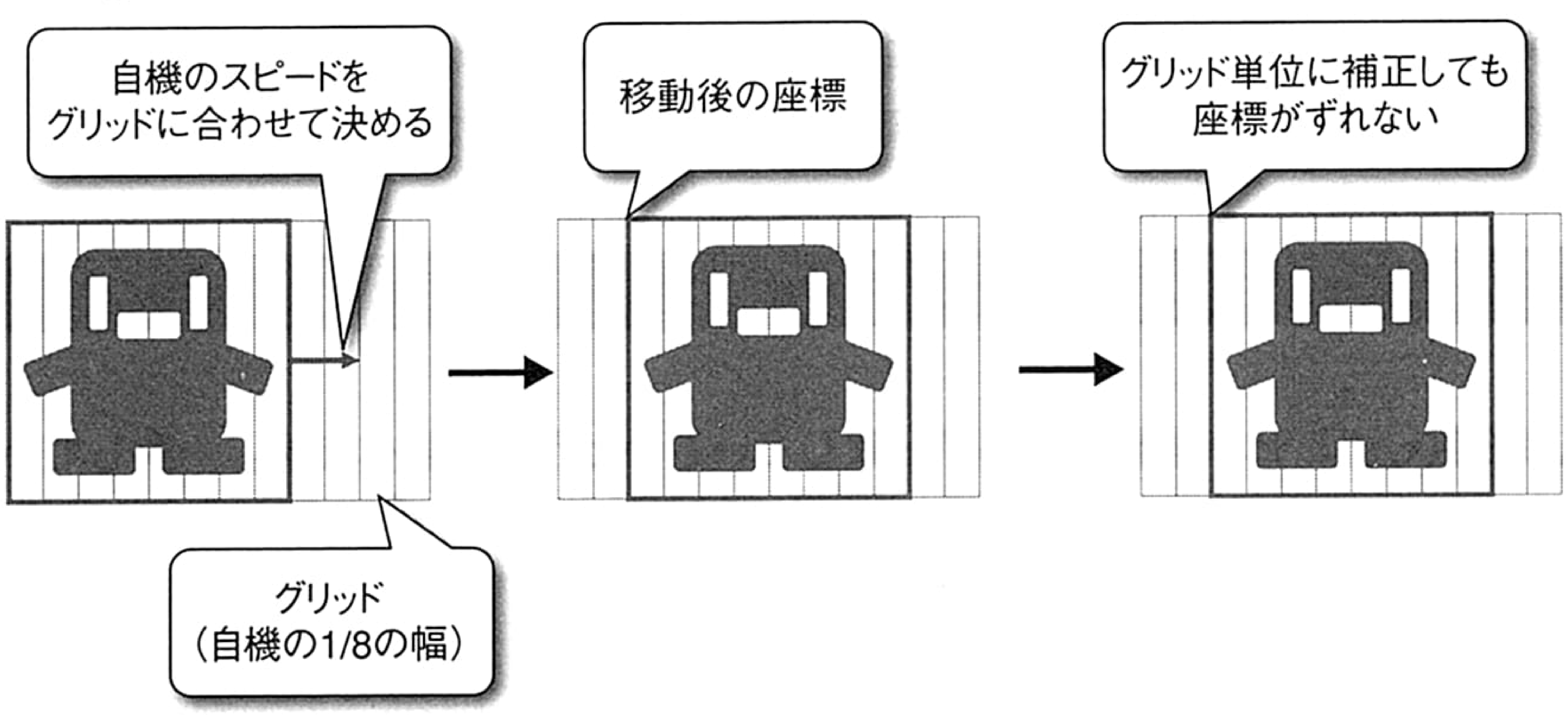
最初に、

```
var GRID=40;
var PLAYER_SPEED=8/GRID;
```

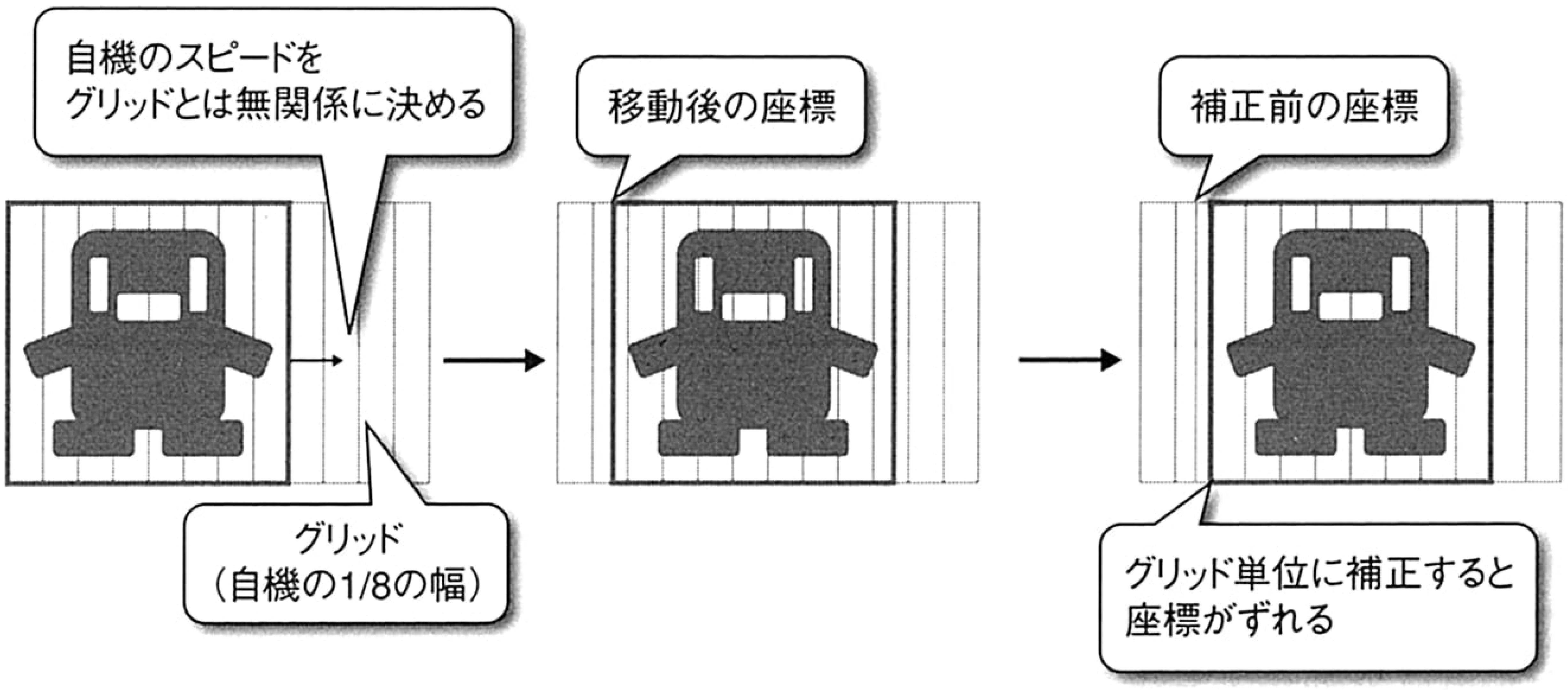
のような定数を定義します。GRIDはグリッドで、PLAYER_SPEEDは自機のスピードです。自機のスピードは1/5（8/40)、つまり単位時間（50ミリ秒）あたりにセル1/5個分だけ移動します。

グリッドの値を使って自機のスピードを決めるのは、座標を補正したときに、自機の位置がずれないようにするためです(Fig.29)。グリッドと無関係に自機のスピードを決めてしまうと、補正前と補正後の座標が大きく変わってしまい、自機の動きがぎこちなくなることがあります(Fig.30)。

Fig.29: グリッドに合わせてスピードを決める



▼ **Fig.30**: グリッドとは無関係にスピードを決める



このグリッドの値は、Playerオブジェクトにおいて、

```
obj.sprite.setGrid(GRID);
```

のように設定します。これで自機の座標を設定するたびに、座標が1/GRID単位（ここでは1/40単位）に補正されます。

List7では、自機の移動先が壁のセルかどうかを判定するために、

```
function isWall(x, y) {
    return field.getCell(x, y)==CELL_WALL;
}
```

のようなisWall関数を定義しました。FieldオブジェクトのgetCell関数(p.184)を使って、指定された位置のセルを取得し、壁を表す番号(ここではCELL_WALL)と比較します。

自機の移動はPlayerオブジェクトのalive関数で行います。ここではまず、

```
if (x==cx && y==cy) {
```

のように、座標(x, y)とセル座標(cx, cy)が一致しているかどうかを調べます。一致していたら、

```
vx=vy=0;
if (game.isKey(KEY_LEFT)) vx=-1; else
if (game.isKey(KEY_RIGHT)) vx=1; else
if (game.isKey(KEY_UP)) vy=-1; else
if (game.isKey(KEY_DOWN)) vy=1;
```

のように、カーソルキーの入力に応じて移動方向(vx, vy)を決めます。そして、

```
if (isWall(cx+vx, cy+vy)) {
    vx=vy=0;
}
```

のように移動先が壁かどうかを調べて、壁の場合には移動をやめます。移動する場合には、

```
cx+=vx;
cy+=vy;
```

のようにセル座標を更新します。

セル座標とは別に、

```
x+=vx*PLAYER_SPEED;
y+=vy*PLAYER_SPEED;
obj.sprite.setXY(x, y);
```

のように座標(スプライトの座標)も更新します。このとき、

```
game.setViewXY(x-XMAX/2, y-YMAX/2);
```

のようにスクロール座標も設定します。(XMAX, YMAX)は画面のサイズ(セル単位)です。ここでは自機が常に画面の中央にくるように、画面をスクロールさせます。

List7では敵との当たり判定処理も行います。敵に接触したら自機は消滅状態になり、消滅のアニメーションが終わったらゲームオーバーとなります。

List7: フィールドを動き回る自機(mazeChase.js)

```
// グリッド(座標を補正するための定数)
var GRID=40;

// 自機のスピード
var PLAYER_SPEED=8/GRID;

// 壁を表すセルの種類
var CELL_WALL=1;

// 自機の状態(通常・消滅)
```

```
var STATE_PLAYER_ALIVE=0;
var STATE_PLAYER_DEAD=1;

// 指定した座標のセルが壁のときtrueを返す
function isWall(x, y) {
  return field.getCell(x, y)==CELL_WALL;
}

// 自機
function Player(x, y) {
  var obj=this;

  // スプライトの作成
  obj.sprite=game.createSprite();

  // アニメーション・当たり判定・グリッドの設定
  obj.sprite.initAnim(playerWalkAnim);
  obj.sprite.initHit(playerHit);
  obj.sprite.setGrid(GRID);

  // 座標とスクロール座標の設定
  obj.sprite.setXY(x, y);
  game.setViewXY(x-XMAX/2, y-YMAX/2);

  // 状態・速度・セル座標の設定
  obj.sprite.setState(STATE_PLAYER_ALIVE);
  obj.vx=0;
  obj.vy=0;
  obj.cx=x;
  obj.cy=y;

  // 更新処理
  obj.sprite.onUpdate=function() {

    // 状態に応じて分岐する
    switch (obj.sprite.getState()) {
      case STATE_PLAYER_ALIVE: obj.alive(); break;
      case STATE_PLAYER_DEAD: obj.dead(); break;
    }
  }

  // 通常状態
  obj.alive=function() {

    // 座標・速度・セル座標の取得
    var x=obj.sprite.getX(), y=obj.sprite.getY();
    var vx=obj.vx, vy=obj.vy;
```

```
var cx=obj.cx, cy=obj.cy;

// 自機の座標が整数のとき(座標とセル座標が一致しているとき)
if (x==cx && y==cy) {

  // カーソルキーの入力に応じて上下左右に移動する
  vx=vy=0;
  if (game.isKey(KEY_LEFT)) vx=-1; else
  if (game.isKey(KEY_RIGHT)) vx=1; else
  if (game.isKey(KEY_UP)) vy=-1; else
  if (game.isKey(KEY_DOWN)) vy=1;

  // 移動先が壁のときは移動しない
  if (isWall(cx+vx, cy+vy)) {
    vx=vy=0;
  }

  // セル座標の更新
  cx+=vx;
  cy+=vy;
}

// 座標の更新
x+=vx*PLAYER_SPEED;
y+=vy*PLAYER_SPEED;
obj.sprite.setXY(x, y);

// 速度とセル座標の更新
obj.vx=vx;
obj.vy=vy;
obj.cx=cx;
obj.cy=cy;

// 移動中はアニメーションを再生する
if (vx!=0 || vy!=0) {
  obj.sprite.stepAnim();
}

// 自機が画面中央にくるようにスクロール座標を更新する
game.setViewXY(x-XMAX/2, y-YMAX/2);

// 敵との当たり判定処理
if (game.testHitSprite(obj.sprite)) {

  // 敵に接触したら消滅状態にする
  obj.sprite.initAnim(vanishAnim);
  obj.sprite.setState(STATE_PLAYER_DEAD);
```

```
    } else {

      // 敵に接触しない限りはスコアを加算する
      game.addScore(1);
    }
  }

  // 消滅状態
  obj.dead=function() {

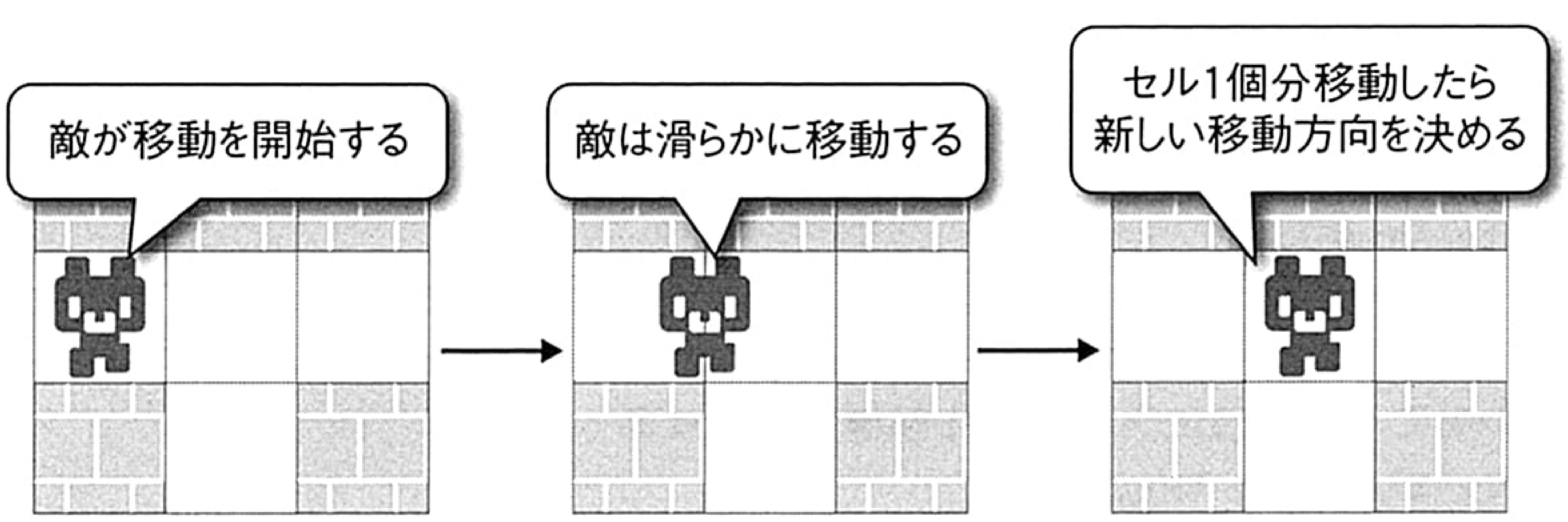
    // アニメーションを再生する
    obj.sprite.stepAnim();

    // アニメーションが終わったらゲームを初期化する
    if (obj.sprite.isAnimOver()) {
      init();
    }
  }
}
```

フィールドを動き回る敵

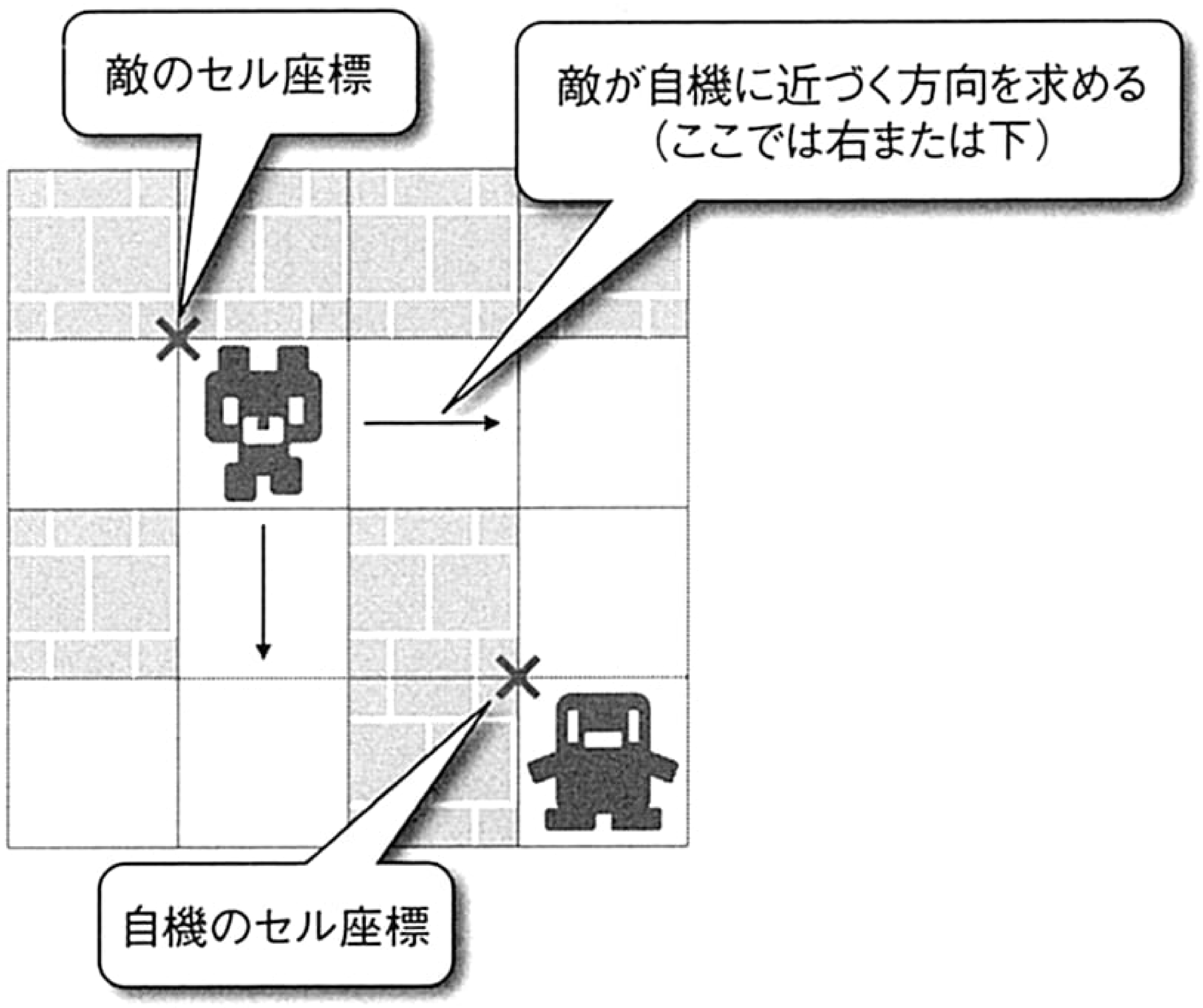
自機に続いて、今度はフィールドを動き回る敵を作ってみましょう。自機と同様に、敵もセル単位で動かします(Fig.31)。

Fig.31: 敵をセル単位で動かす



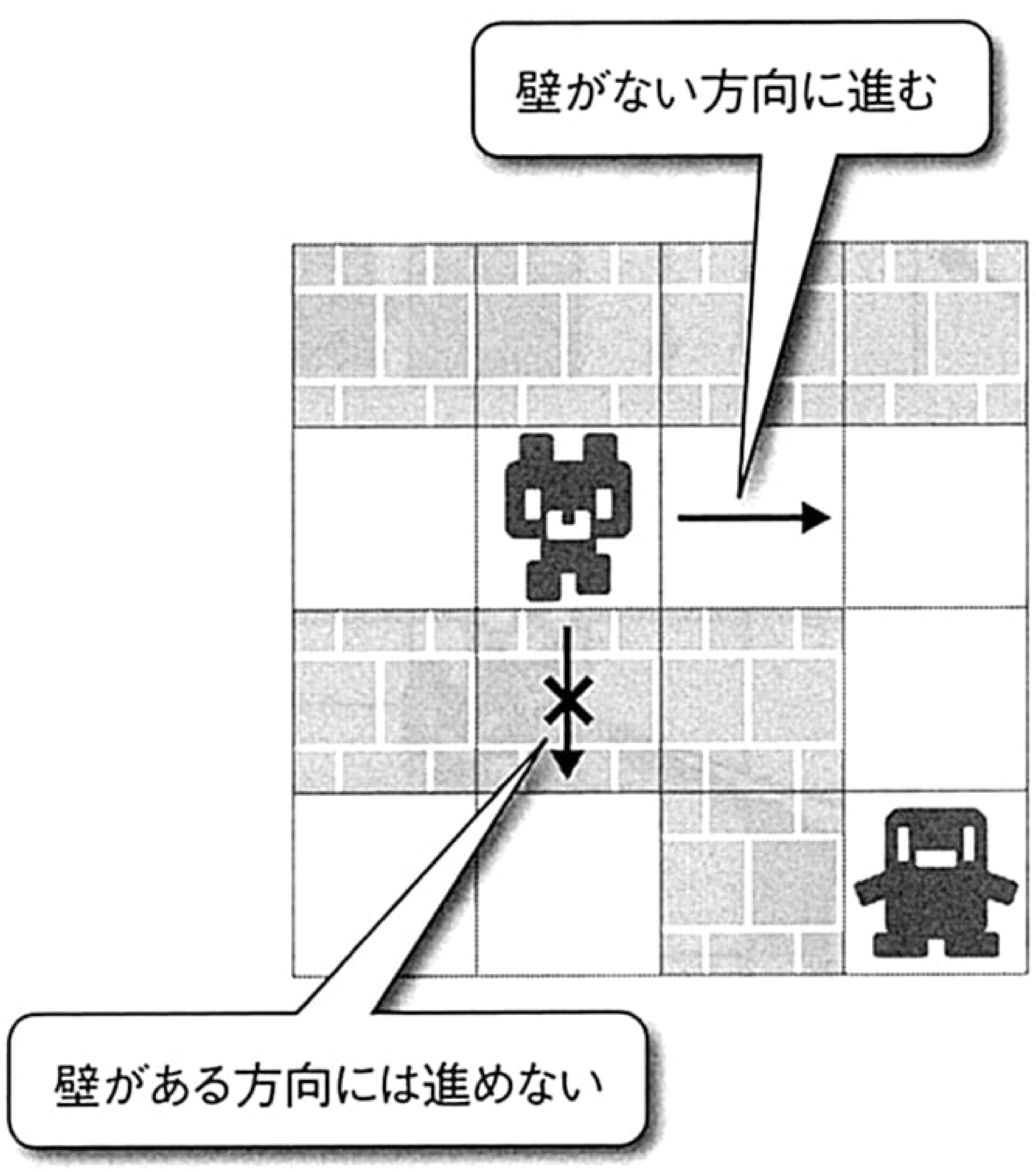
座標とセル座標が一致したときには、次に敵をどちらに動かすかを決めます。敵に自機を追いかけさせるためには、敵の座標を自機の座標に近づける必要があります。そこで、敵と自機のセル座標を調べて、敵が自機に近づく方向を求めます(Fig.32)。座標ではなくセル座標を使うのは、座標の微妙な変化に惑わされて、上手く自機を追いかけられないことがあるためです。

▼ **Fig.32**: 自機に近づく方向を求める



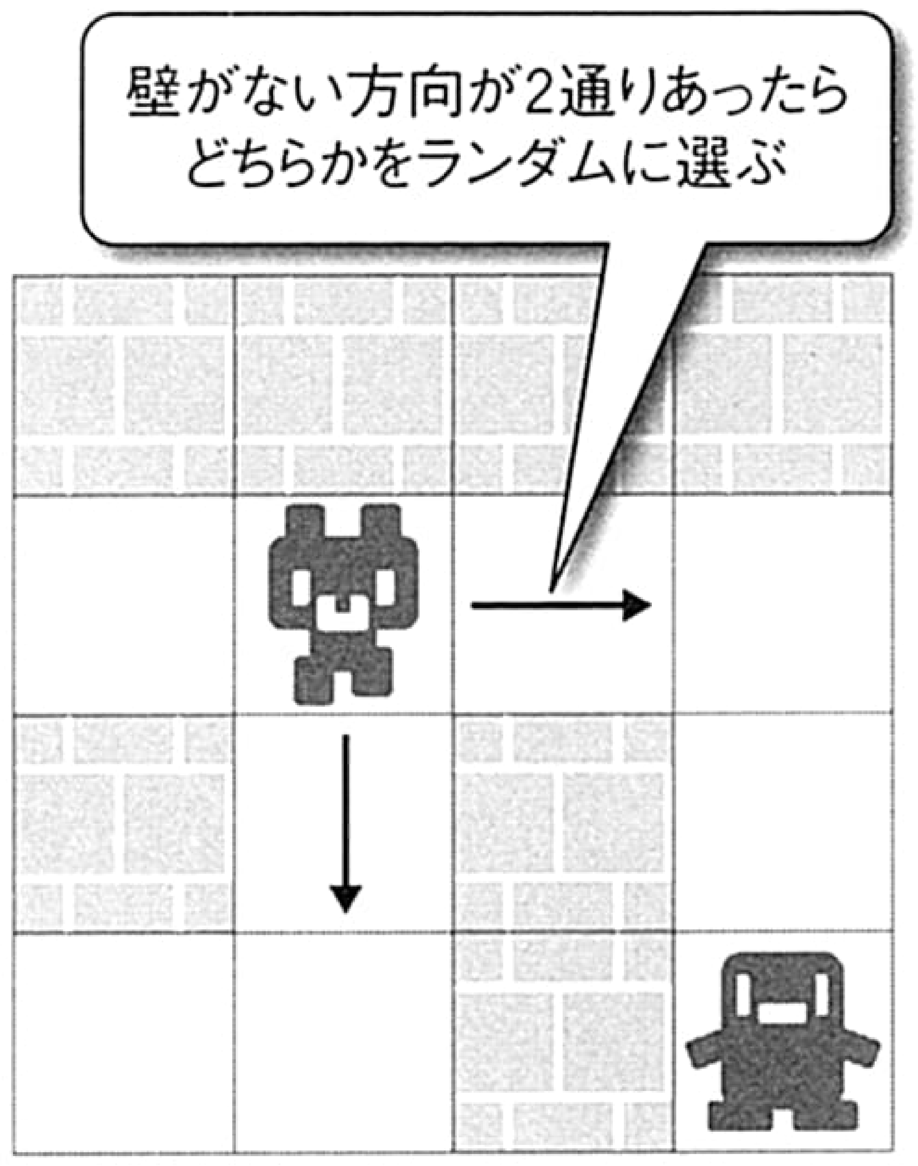
次に、求めた方向に壁があるかどうかを調べます(Fig.33)。壁がある方向には進めません。壁がない方向があれば、その方向に移動します。

▼ **Fig.33**: 壁がない方向に移動する

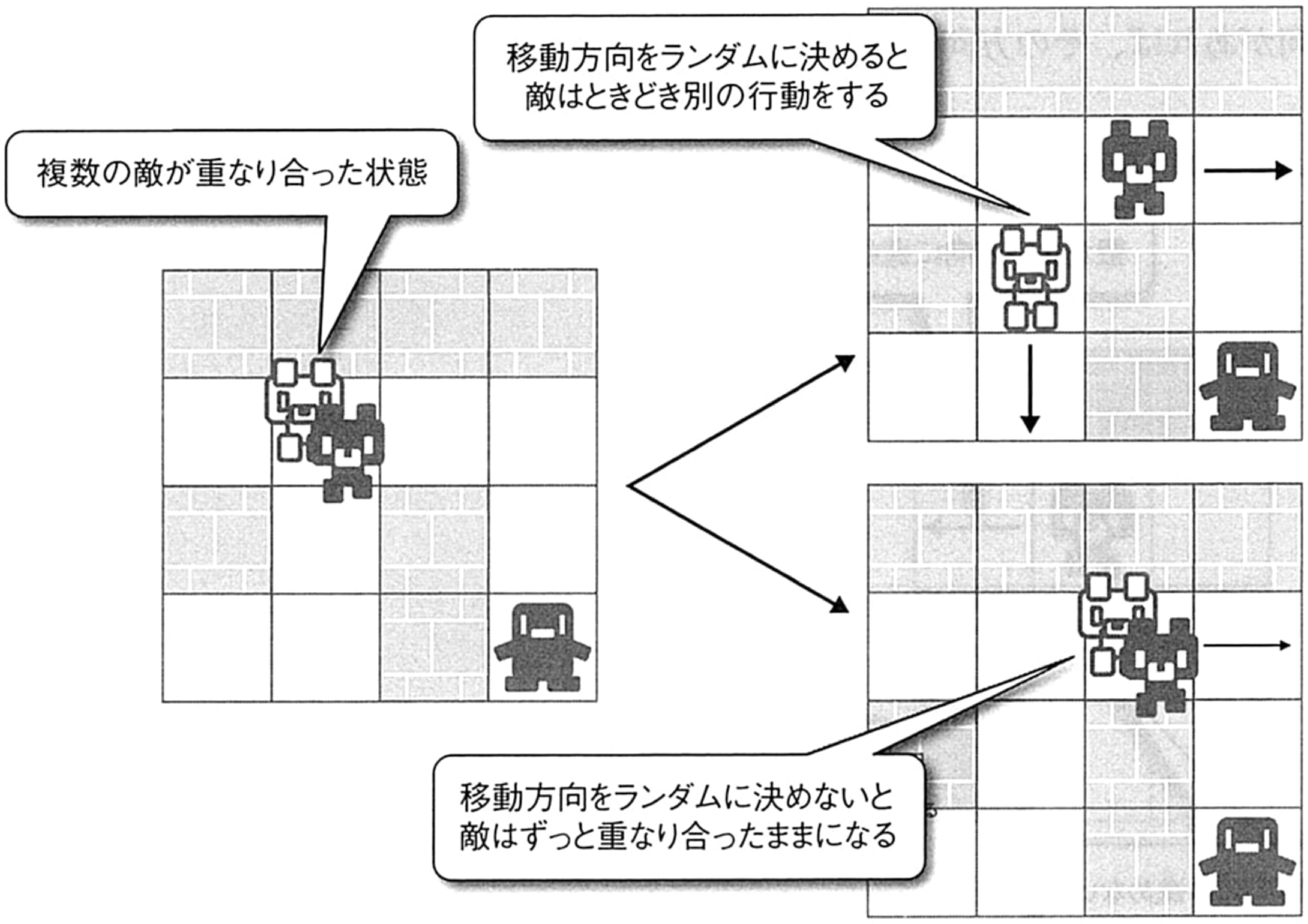


壁がない方向が2通りあるときには、どちらに移動するのかをランダムに決めます(Fig.34)。十字路や三叉路などでは、X方向(横)とY方向(縦)のいずれにも進めることがよくあります。ここで移動方向をランダムに決めることにより、敵の動きに変化がつきます。こういった変化をつけないと、複数の敵が同じ位置に重なり合った場合に、以後全く同じ動きをしてしまいます(Fig.35)。

▼ Fig.34: 移動方向をランダムに決める

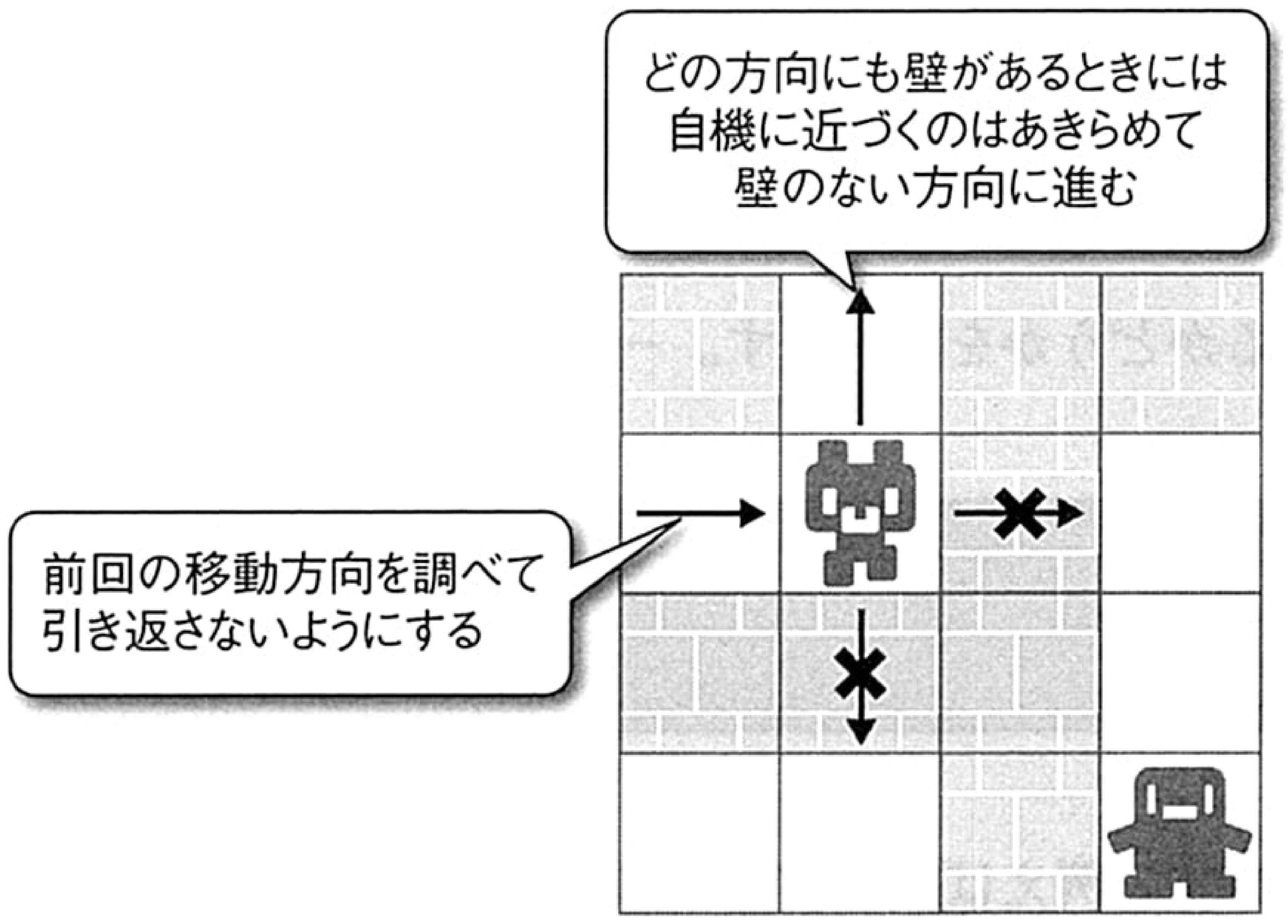


▼ Fig.35: 敵の動きに変化をつける



どの方向にも壁があるときには、自機に近づくのはいったんあきらめて、壁がない方向に進みます(Fig.36)。このとき前回の移動方向を調べて、いま来た道をすぐに引き返さないようにすると、敵の動きが自然になります。

▼ Fig.36: 壁がない方向に進む



フィールドを動き回る敵のプログラム

List8は敵のプログラムです。敵は自機を追いかけるようにフィールド内を動き回ります。

最初に敵のスピードを、

```
var ENEMY_SPEED=5/GRID;
```

のように定義します。自機(p.207)と同様に、グリッドの値(GRID)を使って設定します。自機のスピードは8/GRIDなので、敵はやや遅め(自機の6割程度)のスピードです。このように敵から逃げるゲームは、敵を自機よりも少し遅くしておくと面白くなります。

敵を動かす処理はonUpdate関数で行います。まず、敵のセル座標(cx, cy)を、

```
var cx=Math.floor(x), cy=Math.floor(y);
```

のように計算します。Math.floor関数を使って、座標(x, y)を整数に変換することにより、セル座標を求めます。自機のセル座標(px, py)は、

```
var px=player.cx, py=player.cy;
```

のように、Playerオブジェクトのプロパティから取得します。

次に、求めた敵のセル座標(cx, cy)を使って、

```
if (x==cx && y==cy) {
```

のように、座標とセル座標が一致するかどうかを調べます。一致したら、

```
vx=vy=0;
if (px<cx) vx=-1; else if (px>cx) vx=1;
if (py<cy) vy=-1; else if (py>cy) vy=1;
```

のように、自機に近づく方向を求めます。例えば敵の右上に自機がいるときには、(vx, vy)は(1, －1)になります。敵の左に自機がいるときには、(vx, vy)は(－1, 0)です。

方向が求まったら、

```
if (isWall(cx+vx, cy) || vx==-ovx) vx=0;
if (isWall(cx, cy+vy) || vy==-ovy) vy=0;
```

のように、その方向に進めるかどうかを調べます。壁がある場合や、後戻りする方向(今まで進んでいた方向と逆の方向)になる場合には、vxまたはvyを0にして、その方向には進めないことを示します。壁があるかどうかはisWall関数(p.209のList7)で調べます。

X方向とY方向のどちらにも進める場合には、

```
if (vx!=0 && vy!=0) {
    if (randInt(2)==0) vx=0; else vy=0;
}
```

のように、どちらに進むのかをランダムに決めます。randInt(2)は0または1を同じ確率で発生します(p.77)。

X方向とY方向のどちらにも進めない場合には、

```
if (vx==0 && vy==0) {
    if (!isWall(cx-1, cy) && ovx!=1) vx=-1; else
    if (!isWall(cx+1, cy) && ovx!=-1) vx=1; else
    if (!isWall(cx, cy-1) && ovy!=1) vy=-1; else
    if (!isWall(cx, cy+1) && ovy!=-1) vy=1;
}
```

のように、進める方向を探します。進める方向とは、壁がなく、後戻りしない方向です。ここでは左右上下の順に調べて、進める方向があればその方向に進みます。進める方向をランダムに選んでもよいでしょう。

▼ **List8**: フィールドを動き回る敵(mazeChase.js)

```
// 敵のスピード
var ENEMY_SPEED=5/GRID;

// 敵
function Enemy(x, y) {
  var obj=this;

  // スプライトの作成
  obj.sprite=game.createSprite();

  // アニメーション·当たり判定·グリッドの設定
  obj.sprite.initAnim(enemyWalkAnim);
  obj.sprite.initHit(enemyHit);
  obj.sprite.setGrid(GRID);

  // 座標と速度の設定
  obj.sprite.setXY(x, y);
  obj.vx=0;
  obj.vy=0;

  // 更新処理
  obj.sprite.onUpdate=function() {

    // 座標·速度·古い速度の取得
    var x=obj.sprite.getX(), y=obj.sprite.getY();
    var vx=obj.vx, vy=obj.vy;
    var ovx=obj.vx, ovy=obj.vy;

    // 敵のセル座標·自機のセル座標の取得
    var cx=Math.floor(x), cy=Math.floor(y);
    var px=player.cx, py=player.cy;

    // 敵の座標が整数のとき(座標とセル座標が一致しているとき)
    if (x==cx && y==cy) {

      // 自機に近づく方向を求める
      vx=vy=0;
      if (px<cx) vx=-1; else if (px>cx) vx=1;
      if (py<cy) vy=-1; else if (py>cy) vy=1;
```

```
      // 求めた方向に壁があるときや、
      // 後戻りするときには移動をやめる
      if (isWall(cx+vx, cy) || vx==-ovx) vx=0;
      if (isWall(cx, cy+vy) || vy==-ovy) vy=0;

      // X方向・Y方向のどちらにも移動できるときには、
      // 移動方向をランダムに選ぶ
      if (vx!=0 && vy!=0) {
        if (randInt(2)==0) vx=0; else vy=0;
      } else

      // X方向・Y方向のどちらにも移動できないときには、
      // 壁がないいずれかの方向に移動する
      if (vx==0 && vy==0) {

        // 古い速度(ovx, ovy)を調べて、
        // 来た方向に後戻りしないようにする
        if (!isWall(cx-1, cy) && ovx!=1) vx=-1; else
        if (!isWall(cx+1, cy) && ovx!=-1) vx=1; else
        if (!isWall(cx, cy-1) && ovy!=1) vy=-1; else
        if (!isWall(cx, cy+1) && ovy!=-1) vy=1;
      }
    }

    // 座標と速度の更新
    obj.sprite.setXY(x+vx*ENEMY_SPEED, y+vy*ENEMY_SPEED);
    obj.vx=vx;
    obj.vy=vy;

    // アニメーションの再生
    obj.sprite.stepAnim();
  }
}
```

フィールドを動き回るゲームの本体

List9はフィールドを動き回って敵から逃げるゲームの本体です。他のサンプルと同様に、ゲームの初期化を行うinit関数と、メインルーチンのmain関数を定義します。

init関数ではフィールドを初期化し、自機や敵を作成します。まず、

```
field=game.createField();
```

のように、createField関数(p.189)を使ってフィールドを作成します。次に、

```
field.initCellImage(new Array(
    "void.png", "wall.png"
));
```

のようにinitCellImage関数(p.196)を呼び出して、セルの種類に対応する画像ファイルの一覧を登録します。ここでは0番のセルは空間(void.png)に、1番のセルは壁(wall.png)にしました。

続いて、

```
field.initCell(new Array(
    "0000000000000000000000000",
    ...
));
```

のようにinitCell関数を使って、セルを初期化します。文字列の0は空間、1は壁を表します。このサンプルでは迷路のような地形を作りました。

フィールドの周囲に広い余白部分があるのは、スクロールしたときに反対側が表示されないようにするためです。フィールドの反対側が見えるようにしたり、反対側に移動できるようにしてもよいでしょう。

自機と敵は、

```
player=new Player(11, 7);
enemy=new Array();
enemy[0]=new Enemy(6, 3);
enemy[1]=new Enemy(18, 3);
enemy[2]=new Enemy(6, 13);
enemy[3]=new Enemy(18, 13);
```

のように生成します。引数は自機や敵の座標(フィールド上の座標)です。フィールドのデザイン(壁などの配置)を変えたときには、これらの座標も調整する必要があります。第9章では、自機や敵の座標をセルのデータに含める方法を紹介します(p.270)。

▼ **List9**: ゲーム本体(mazeChase.js)

```
// ゲーム本体・自機・敵・フィールドのオブジェクト
var game;
var player;
var enemy;
var field;

// 自機・敵・消滅のアニメーション
var playerWalkAnim;
var enemyWalkAnim;
var vanishAnim;

// 自機・敵の当たり判定
var playerHit;
var enemyHit;

// ゲームの初期化
function init() {

  // 画面の消去
  game.clear();

  // スコアの初期化
  game.setScore(0);

  // フィールドの作成
  field=game.createField();

  // セルの種類に対応する画像の一覧を配列で設定する
  field.initCellImage(new Array(
    "void.png", "wall.png"
  ));

  // セルの初期化
  field.initCell(new Array(
    "0000000000000000000000000",
    "0000000000000000000000000",
    "0000011111111111111100000",
    "0000010000001000000100000",
    "0000010111100011110100000",
    "0000010000101010000100000",
    "0000010110101010110100000",
```

```
    "0000010100000000010100000",
    "0000010001011101000100000",
    "0000010100000000010100000",
    "0000010110101010110100000",
    "0000010000101010000100000",
    "0000010111100011110100000",
    "0000010000001000000100000",
    "0000011111111111111100000",
    "0000000000000000000000000",
    "0000000000000000000000000"
  ));

  // 自機の作成
  player=new Player(11, 7);

  // 敵の作成
  enemy=new Array();
  enemy[0]=new Enemy(6, 3);
  enemy[1]=new Enemy(18, 3);
  enemy[2]=new Enemy(6, 13);
  enemy[3]=new Enemy(18, 13);
}

// メインルーチン
function main() {

  // ゲーム本体の作成
  game=new Game();

  // アニメーションの作成
  playerWalkAnim=new Array(
    "pcWalk0.png", "pcWalk2.png", "pcWalk0.png",
    "pcWalk1.png", "pcWalk3.png", "pcWalk1.png"
  );
  enemyWalkAnim=new Array(
    "enemyWalk0.png", "enemyWalk2.png", "enemyWalk0.png",
    "enemyWalk1.png", "enemyWalk3.png", "enemyWalk1.png"
  );
  vanishAnim=new Array(
    "void.png", "vanish8.png", "vanish6.png",
    "vanish4.png", "vanish2.png", "vanish0.png",
    "vanish1.png", "vanish2.png", "vanish3.png",
    "vanish4.png", "vanish5.png", "vanish6.png",
    "vanish7.png", "vanish8.png", "void.png"
  );

  // 当たり判定の作成
```

```
  playerHit=new Array(0.25, 0.25, 0.75, 0.75);
  enemyHit=new Array(0.25, 0.25, 0.75, 0.75);

  // ゲームの初期化
  init();
}
```

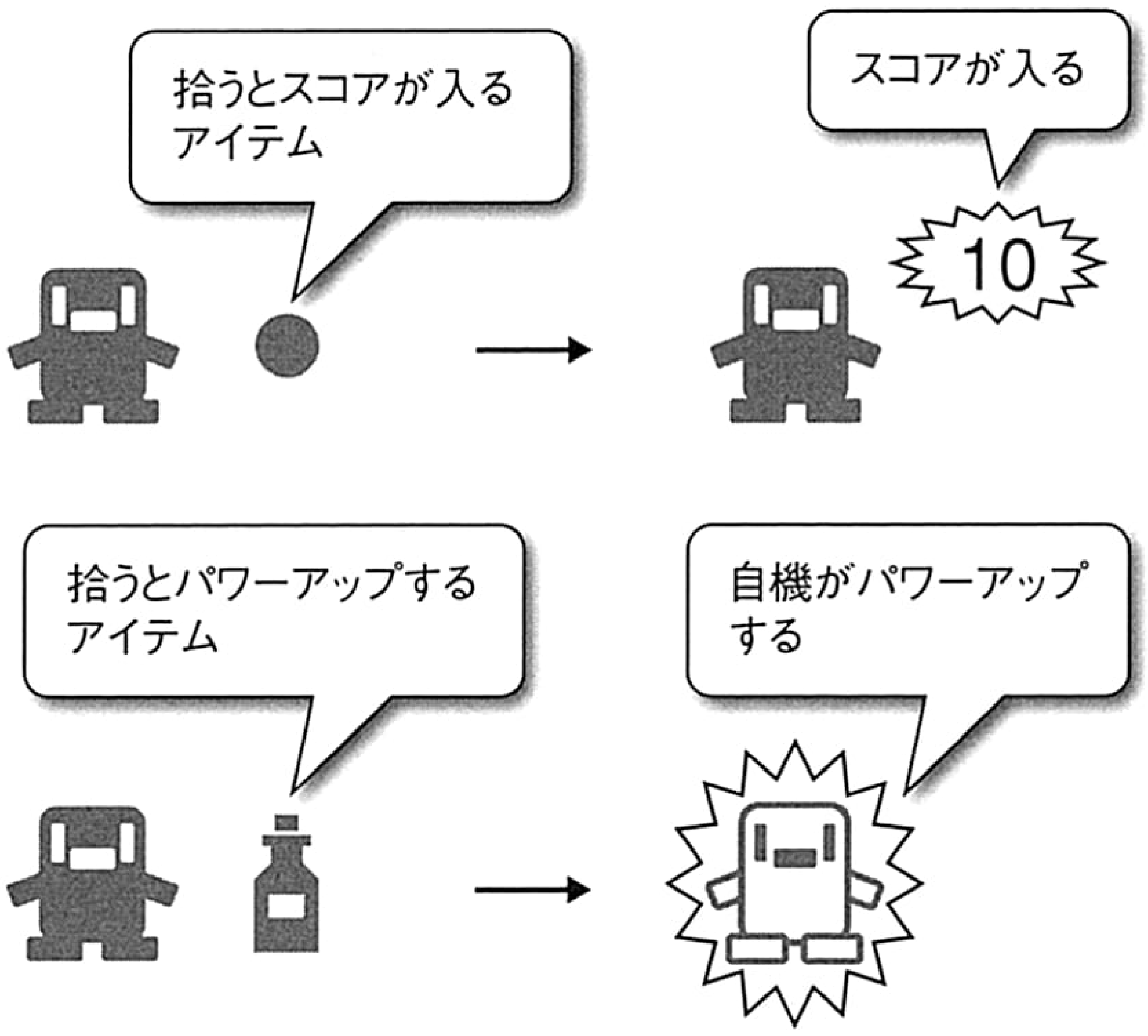
Chapter 8

アイテム

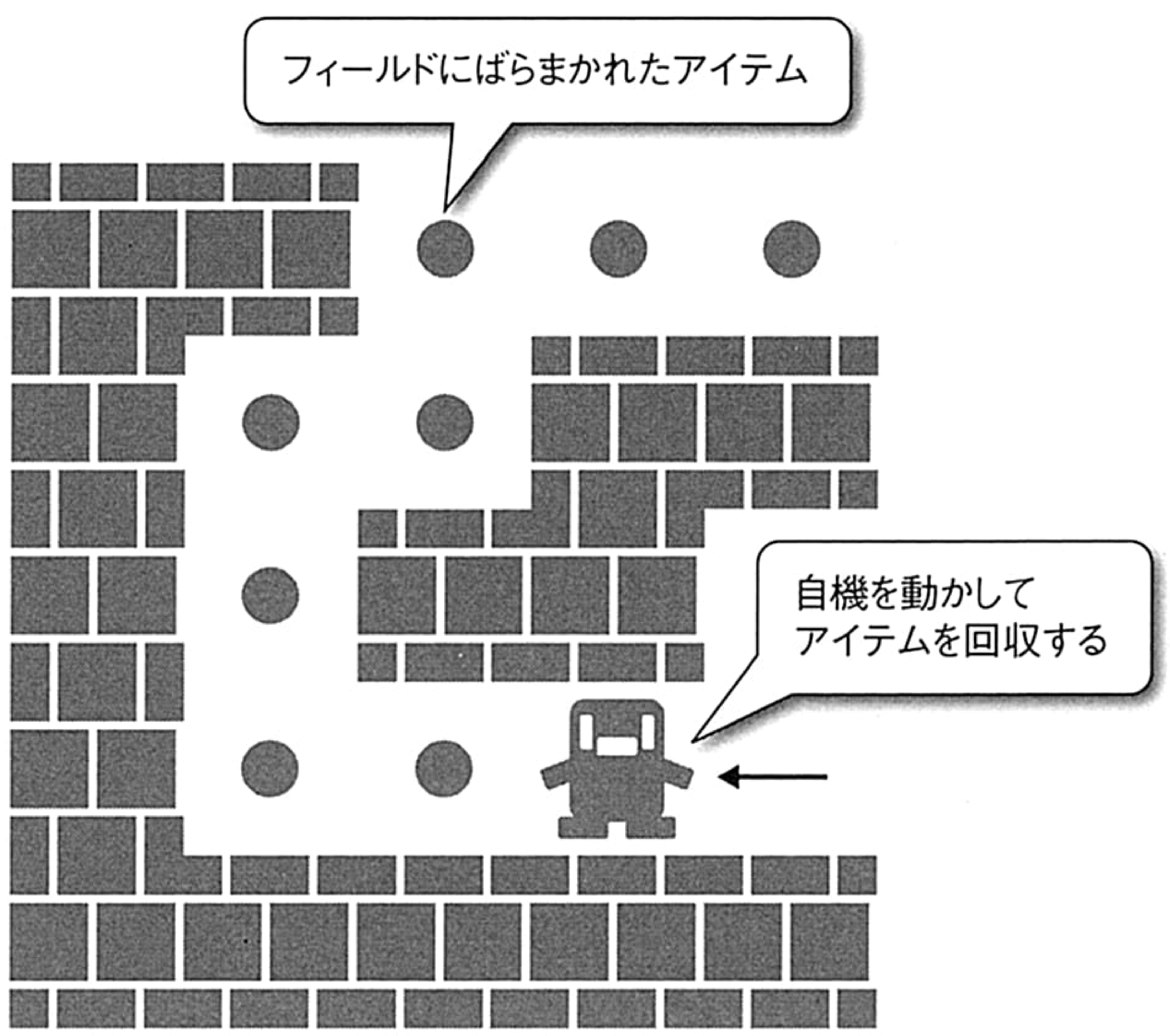
アイテムを拾ってパワーアップ

ゲームには色々なアイテムが出現します。拾うとスコアが入るアイテムや、自機がパワーアップするアイテムなどが代表例です(Fig.1)。多くのアイテムは拾っても拾わなくても構いませんが、「パックマン」のように、フィールドにばらまかれたアイテム(エサ)を集めることが目的のゲームもあります(Fig.2)。

▼ **Fig.1**: 色々なアイテム



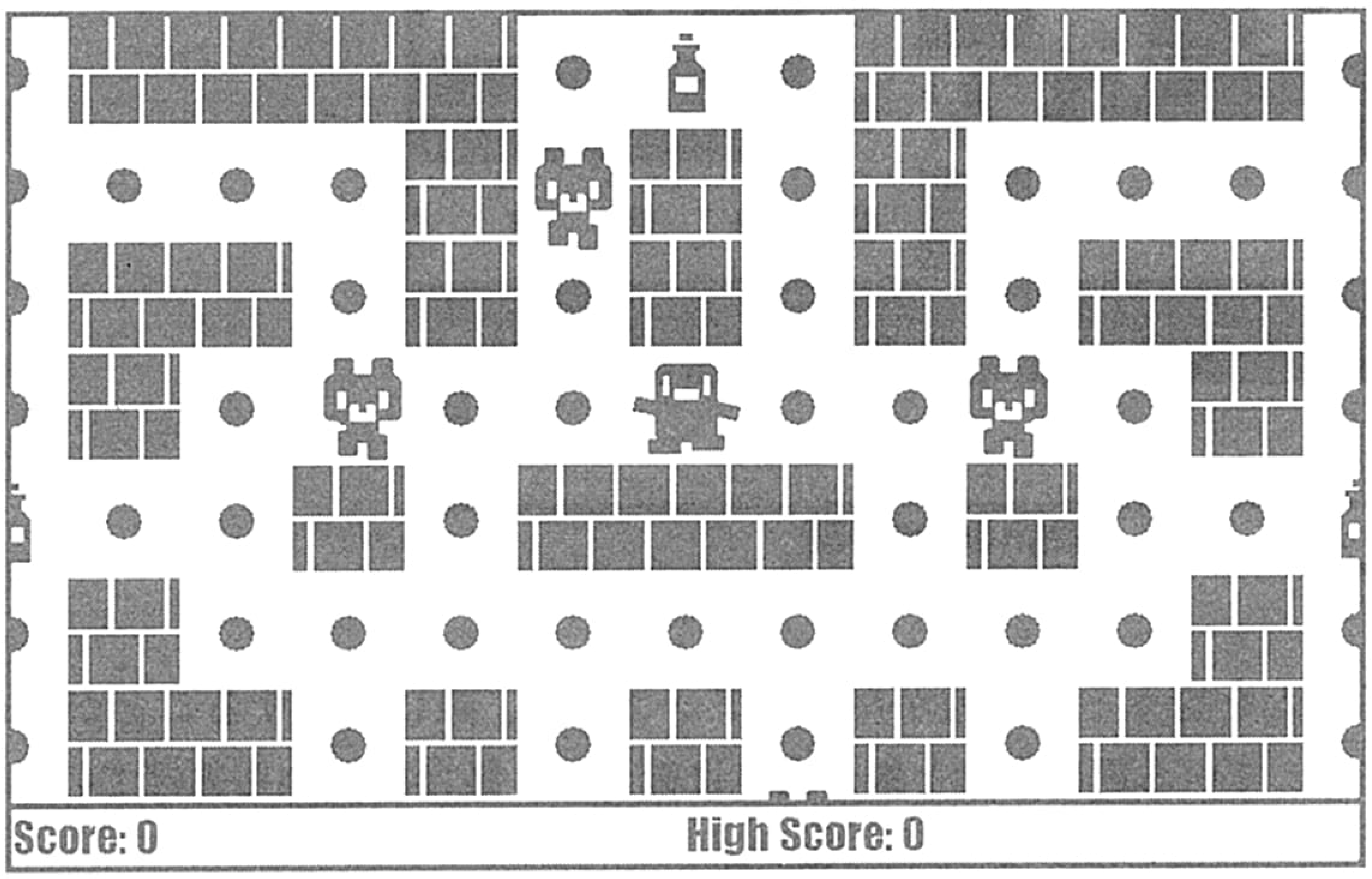
▼ **Fig.2**: アイテムを集める



アイテムがあるとゲームの幅が広がります。そこで本章では、敵から逃げ回る第7章のサンプルに、アイテムを追加してみました。アイテムを拾い集めたり、アイテムでパワーアップしたりといったアクションが楽しめます。ゲームを実行するには、mazeItem.htmlをブラウザで開きます。

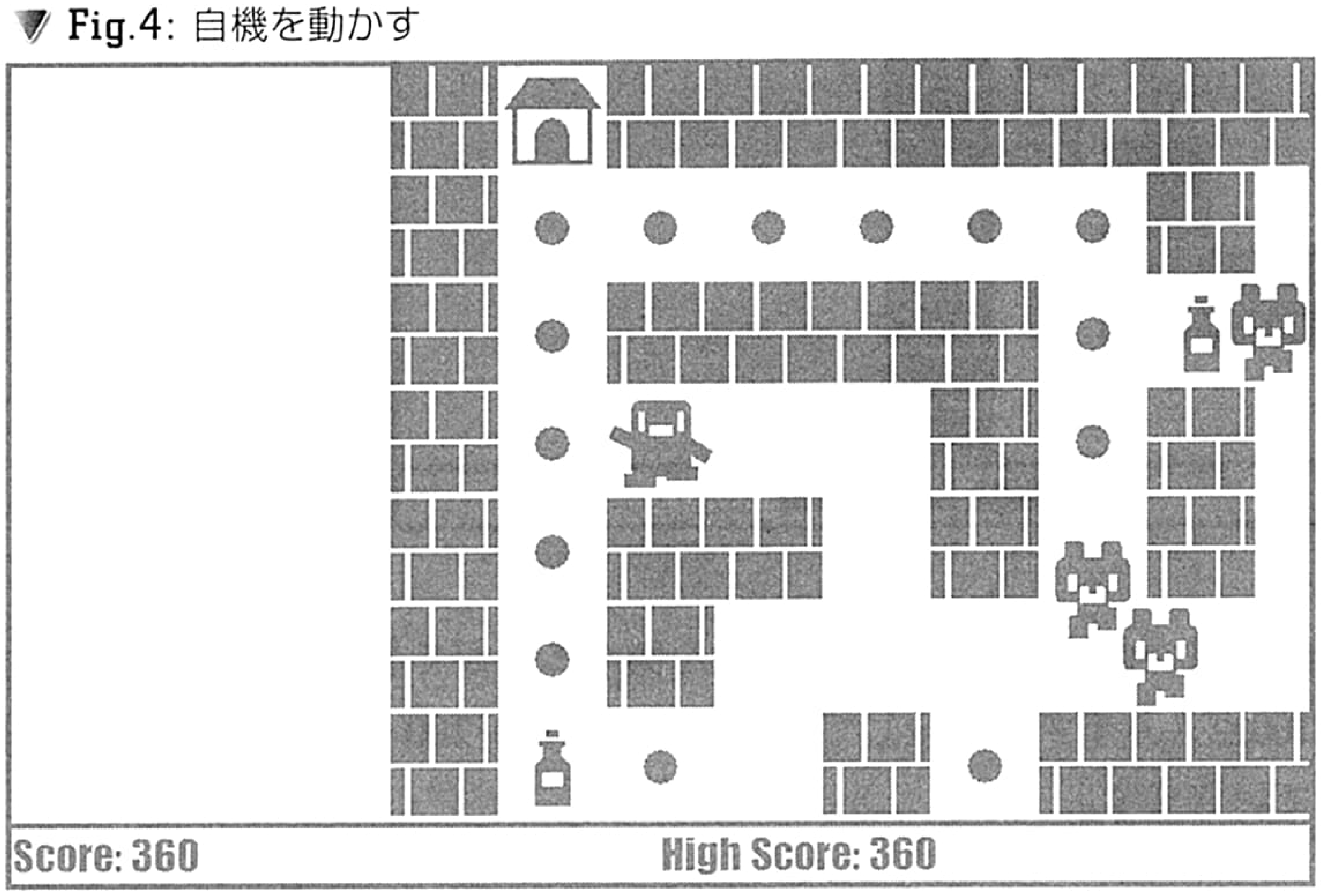
ゲームが始まると、自機がフィールドの中央に出現します(Fig.3)。フィールド中にばらまかれている丸いエサアイテムを全て回収することが、ゲームの目的です。

Fig.3: アイテムを拾うゲーム



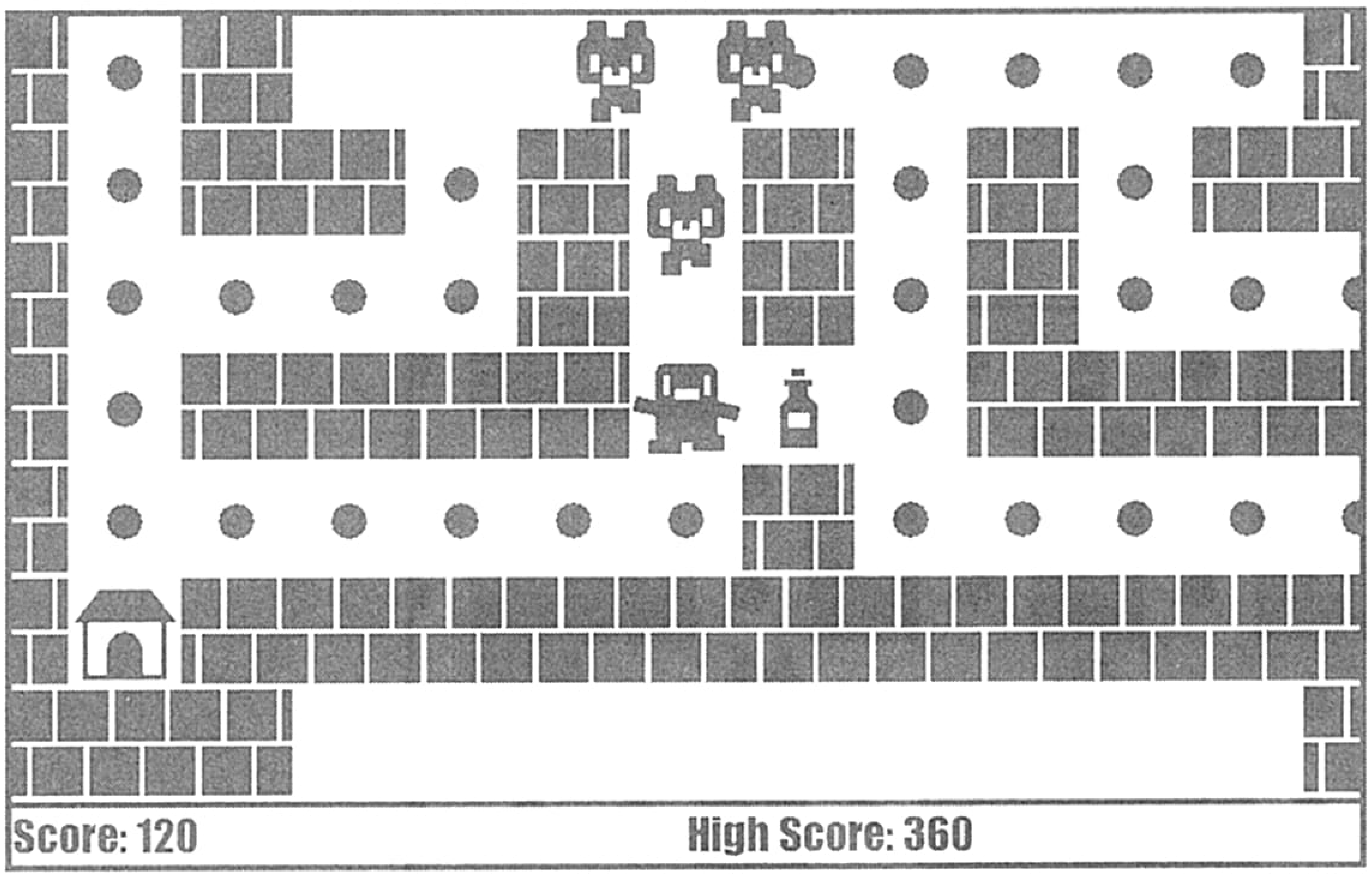
自機はカーソルキーで上下左右に動きます(Fig.4)。自機が通過すると、エサアイテムを拾うことができます。拾ったエサアイテムは消えて、空白の通路になります。エサアイテムを1個拾うごとに、10点のスコアが得られます。

Fig.4: 自機を動かす



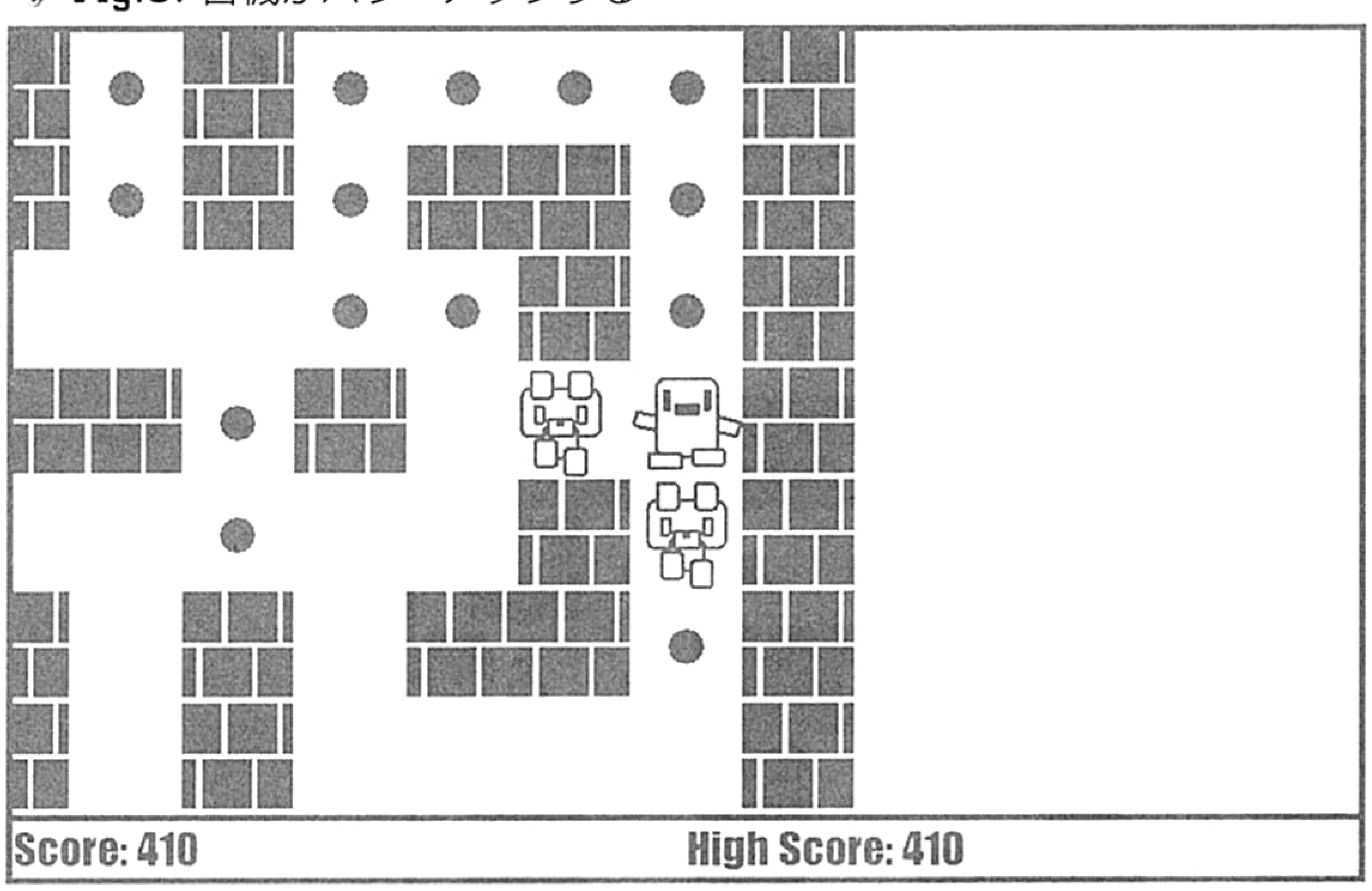
ところどころにビンの形をしたパワーアップアイテムが落ちています(Fig.5)。敵に囲まれて逃げられなくなったら、パワーアップアイテムを拾ってください。

▼ Fig.5: パワーアップアイテム



パワーアップアイテムを拾うと、自機がパワーアップして、敵に逆襲することができます(Fig.6)。パワーアップしている間は自機と敵が白色に変わり、敵は自機から逃げていきます。

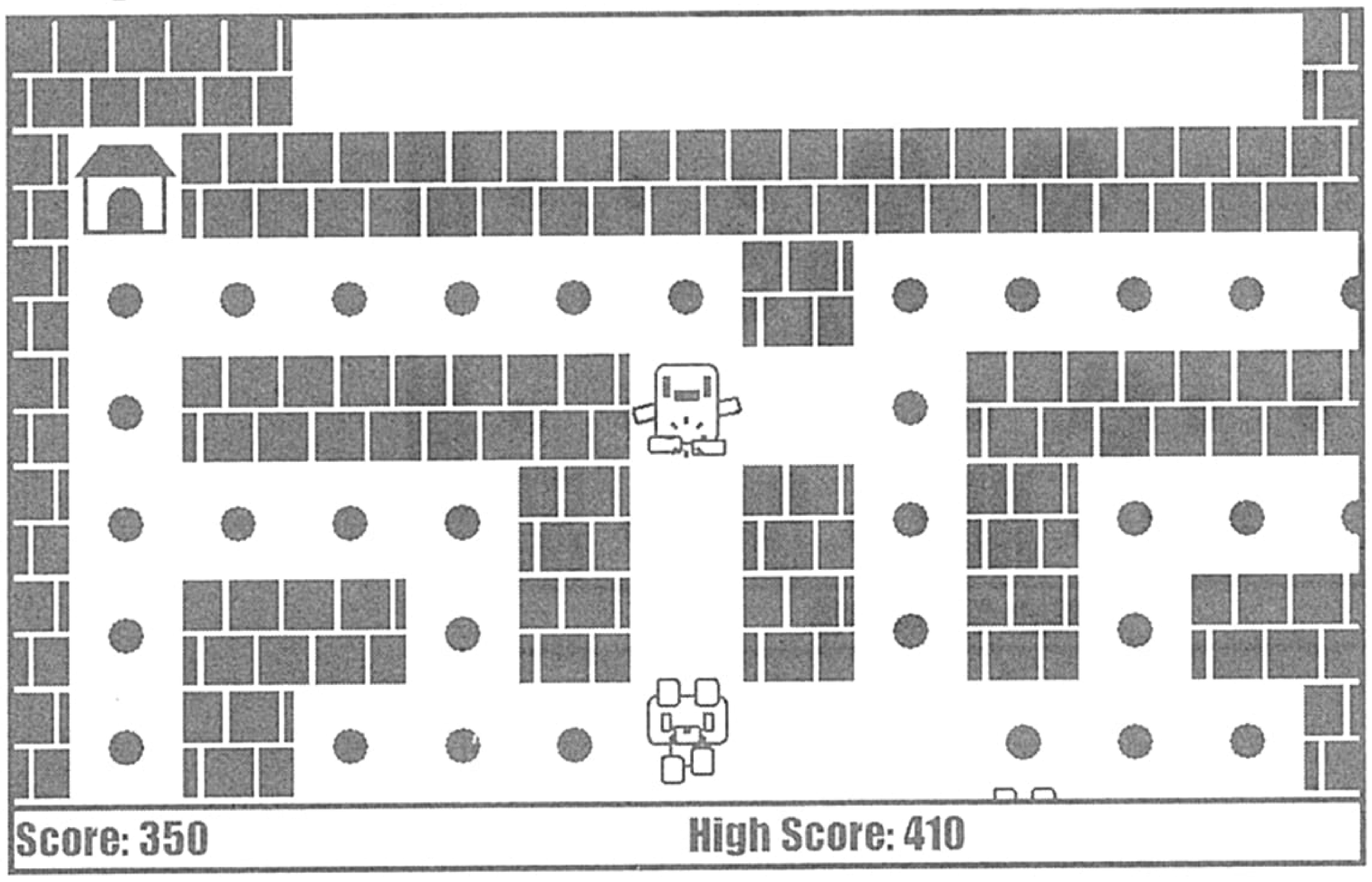
▼ Fig.6: 自機がパワーアップする



パワーアップ中に敵を追いかけて接触すると、敵を倒すことができます(Fig.7)。倒した敵は消滅して、スコアが100点入ります。連続して敵を倒すと、100･200･400･800のように、得られるスコアが高くなっていきます。

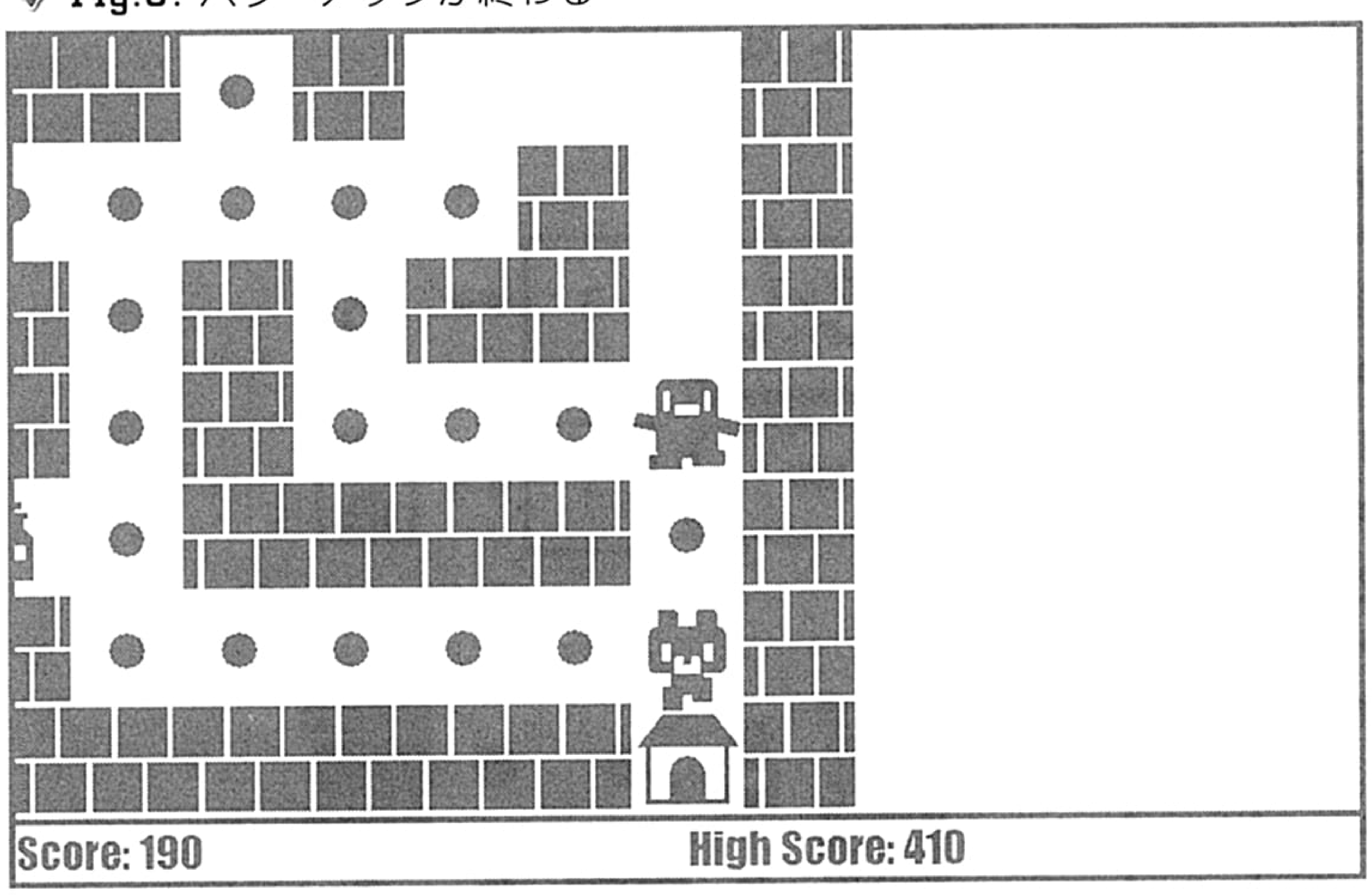
パワーアップは一定時間が経過すると終わります。終了間際になると自機と敵が点滅するので、点滅が始まったら深追いは禁物です。

▼ **Fig.7**: 敵を倒す



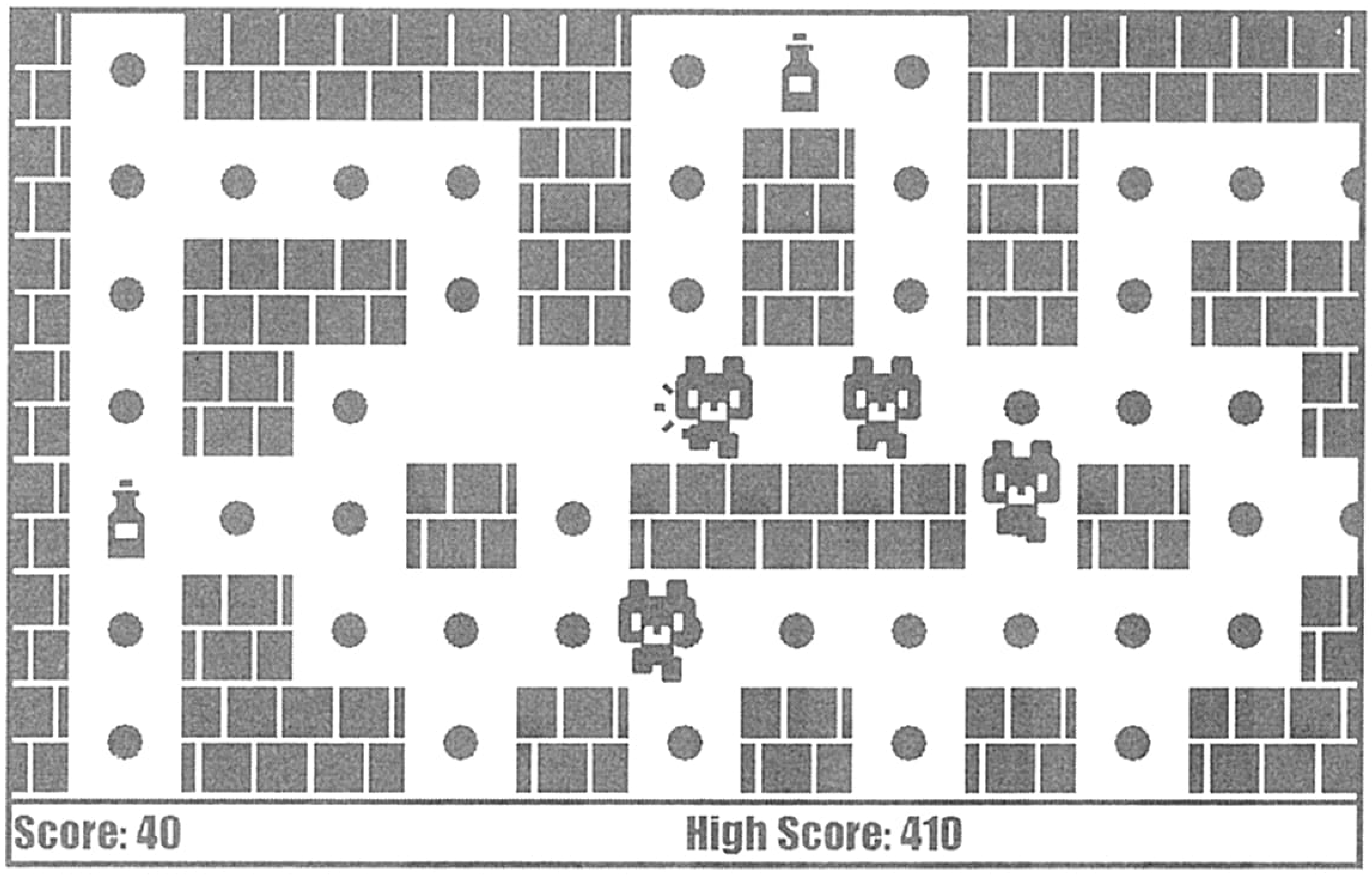
パワーアップが終わると、自機と敵は黒色に戻ります(Fig.8)。倒した敵はフィールドの四隅にある巣から再び出現します。

▼ **Fig.8**: パワーアップが終わる



敵につかまると、自機が消滅してゲームオーバーです(Fig.9)。敵につかまらないように全てのエサアイテムを集めると、ステージクリアになります。パワーアップアイテムは全て拾わなくても構いません。

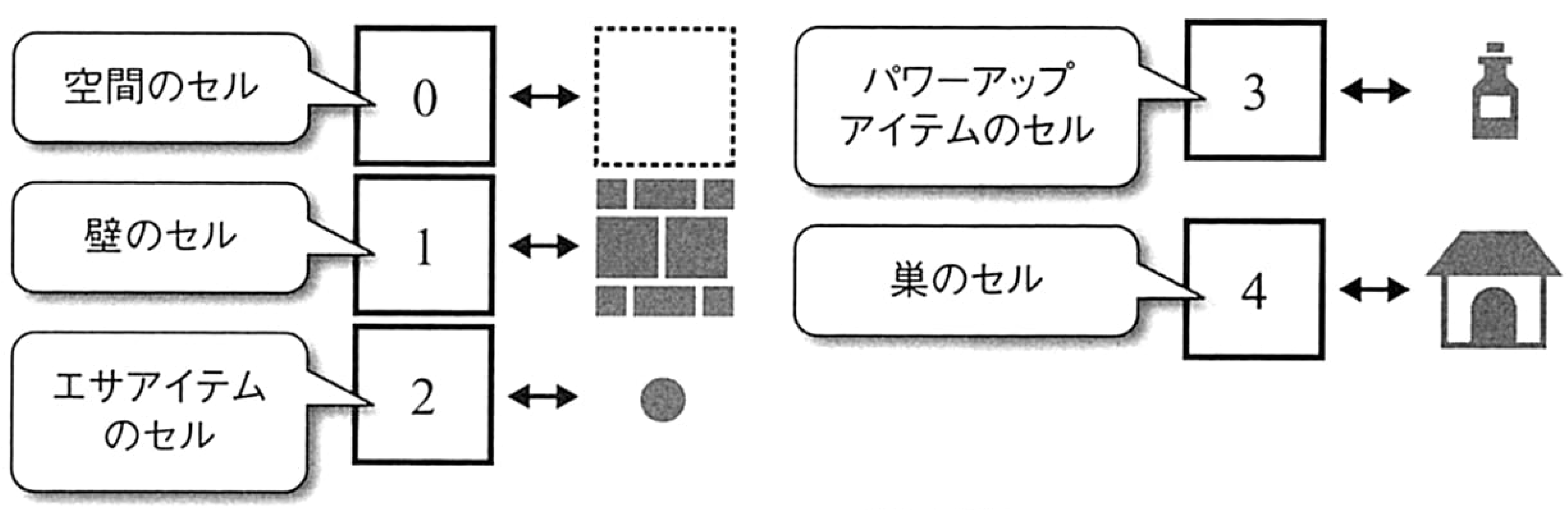
▼ Fig.9：敵につかまるとゲームオーバー



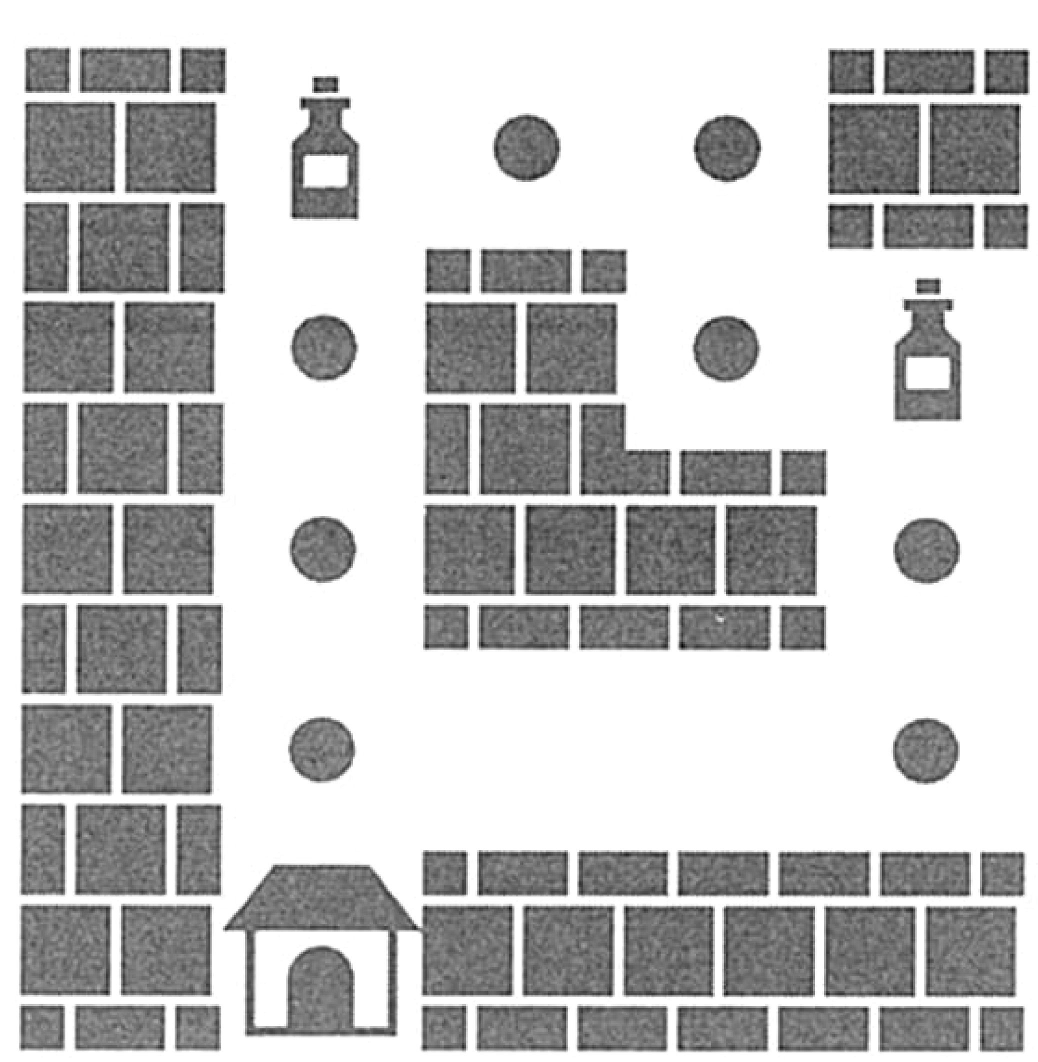
アイテムを拾う自機

自機はフィールドを動き回って、エサアイテムやパワーアップアイテムを拾います。これらのアイテムは、壁と同じようにセルで表します(Fig.10)。セルの種類は0から4までで、それぞれ空間･壁･エサアイテム･パワーアップアイテム･巣に対応します。

▼ Fig.10：アイテムをセルで表現する

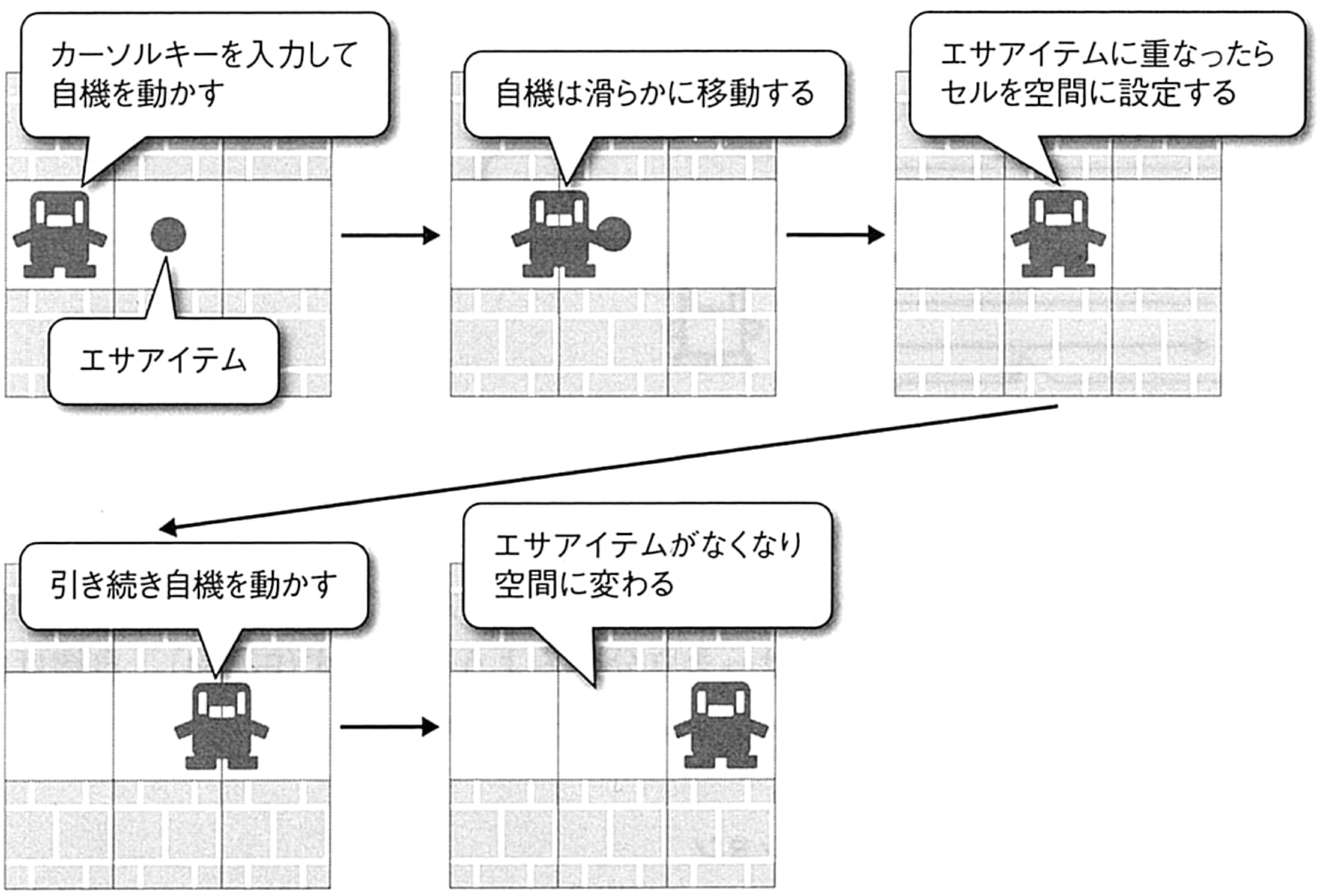


1	3	2	2	1
1	2	1	2	3
1	2	1	1	2
1	2	0	0	2
1	4	1	1	1



第7章と同様に、自機はセル単位で移動します。移動したセルがエサアイテムだったときには、アイテムを拾います(Fig.11)。スコアを加算するとともに、セルを空間(0)に設定して、拾ったアイテムを消します。パワーアップアイテムを拾うときも同様です。

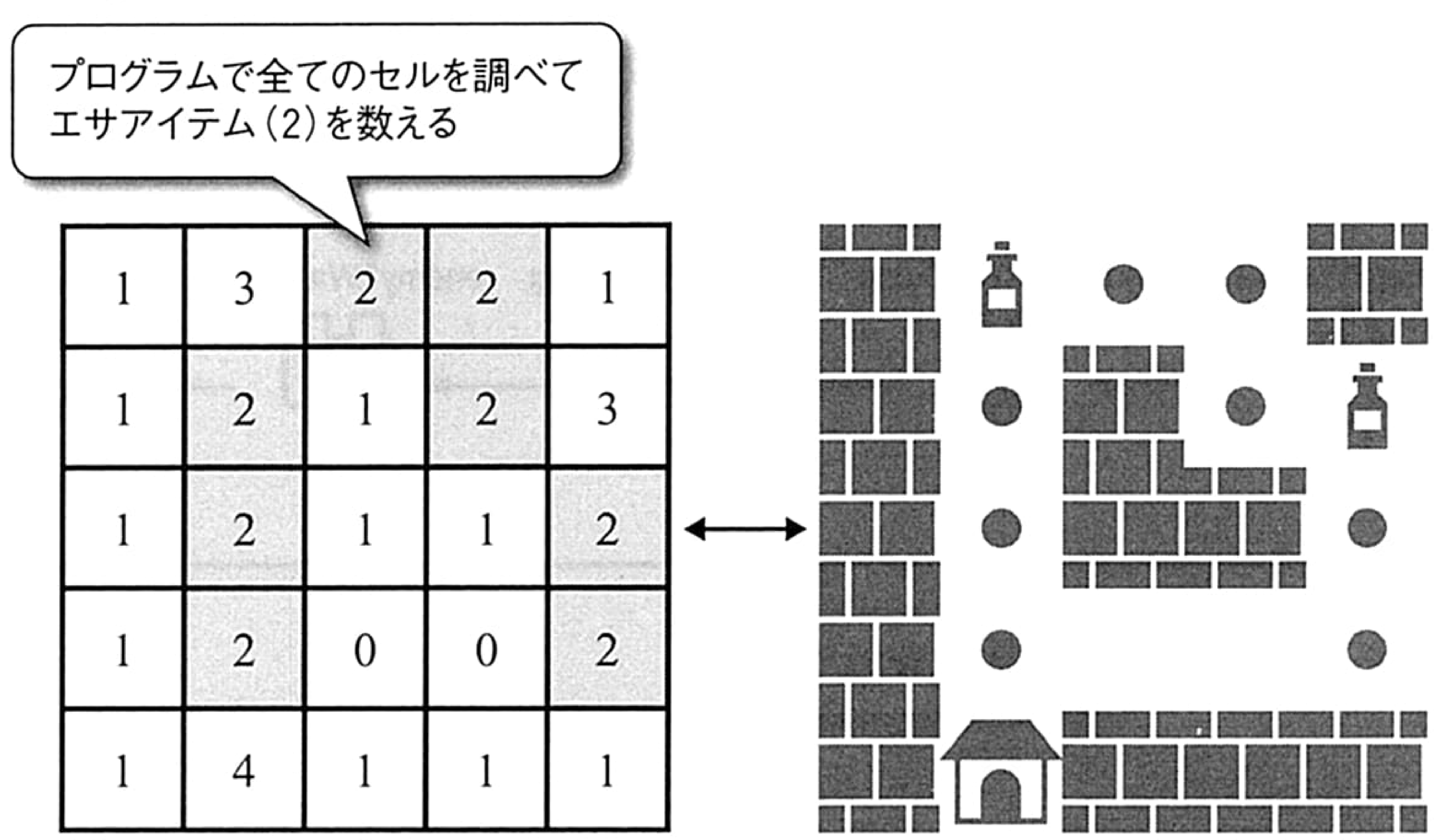
▼ **Fig.11**: 自機がアイテムを拾う



フィールドにある全てのエサアイテムを拾うと、ステージクリアになります。全てのエサアイテムを拾ったかどうかを調べるには、最初にエサアイテムを数えておき、エサアイテムを拾うたびに数を減らして、残りが0になったら終わりとすればよいでしょう。

エサアイテムは数が多いので、手作業で数えるのは大変ですし、間違うこともあります。プログラムでフィールドのセルを調べて、自動的にエサアイテムを数えるとよいでしょう(Fig.12)。

▼ **Fig.12**: エサアイテムを数える

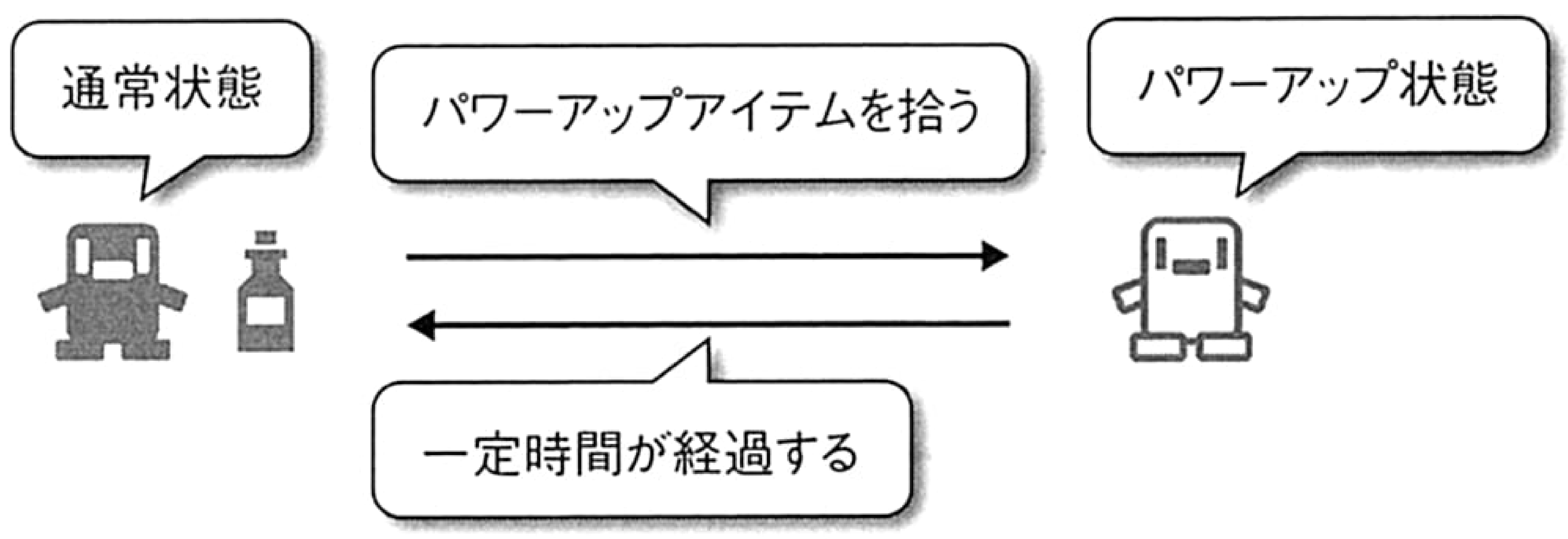


1	3	2	2	1
1	2	1	2	3
1	2	1	1	2
1	2	0	0	2
1	4	1	1	1

自機のパワーアップ

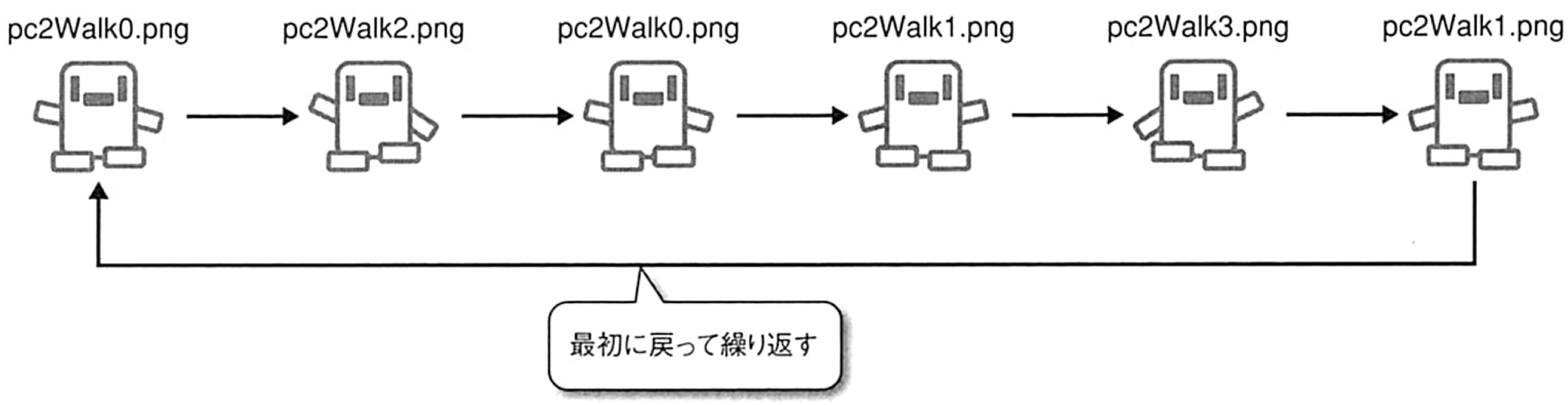
パワーアップアイテムを拾うと、自機は一定時間パワーアップします。パワーアップ中かどうかを区別するために、自機には通常状態とパワーアップ状態があることにしましょう(Fig.13)。パワーアップアイテムを拾ったら、自機をパワーアップ状態にします。

▼ Fig.13: 通常状態とパワーアップ状態



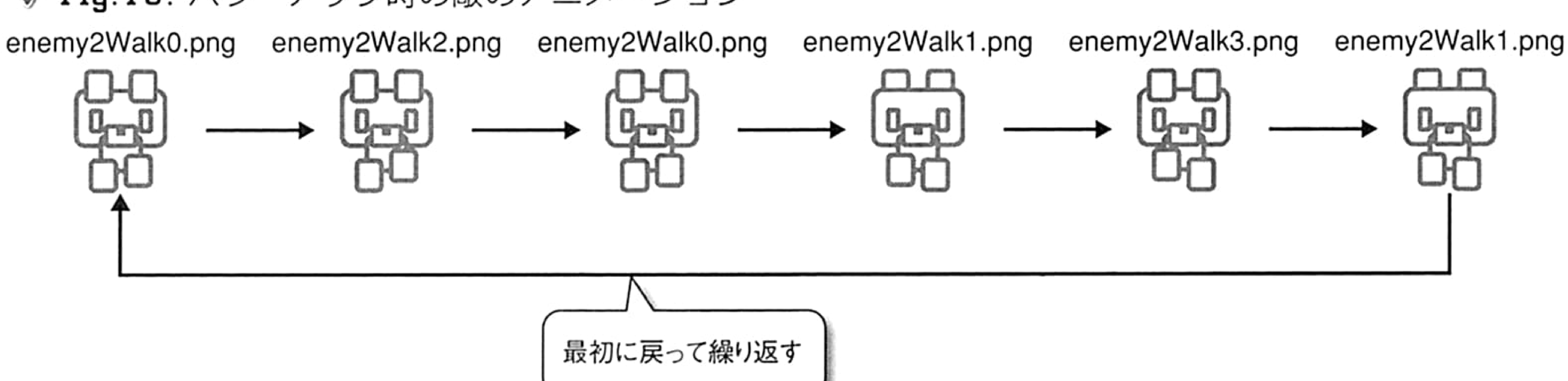
パワーアップ状態になったら、自機のアニメーションをパワーアップ時のものに切り替えて、自機がパワーアップしたことが画面でもはっきり分かるようにします(Fig.14)。自機は白色になりますが、構成は通常時の歩くアニメーションとほぼ同じです。

▼ Fig.14: パワーアップ時の自機のアニメーション



パワーアップしたことをより分かりやすくするために、敵のアニメーションも切り替えます(Fig.15)。こちらも白色になりますが、構成は通常時の敵のアニメーションと同じです。

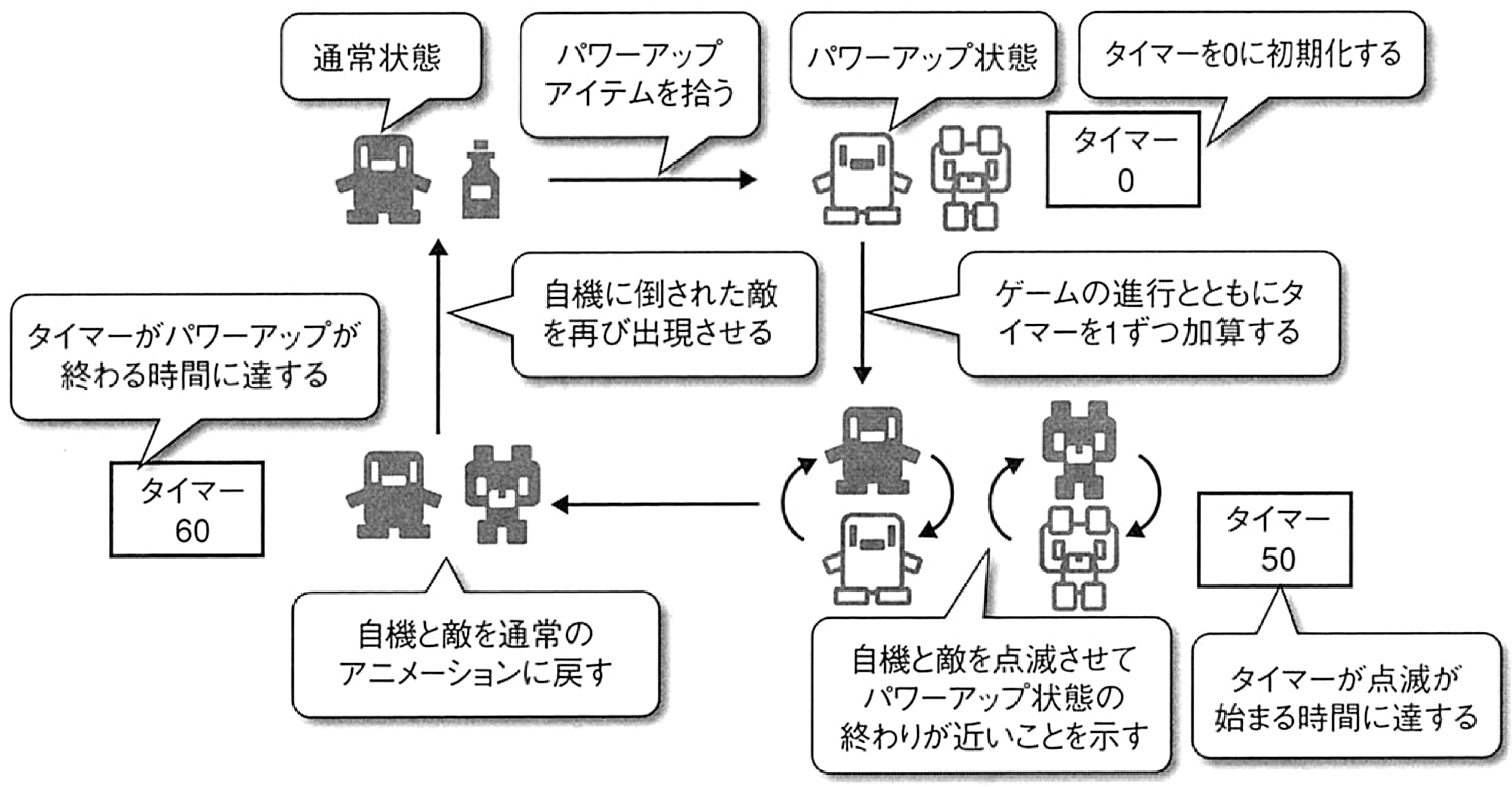
▼ Fig.15: パワーアップ時の敵のアニメーション



さて、パワーアップしてから一定時間が経過すると、パワーアップの終わりが近いことを示す点滅が始まります。さらに一定時間が経過すると、パワーアップが終わり、通常状態に戻ります。

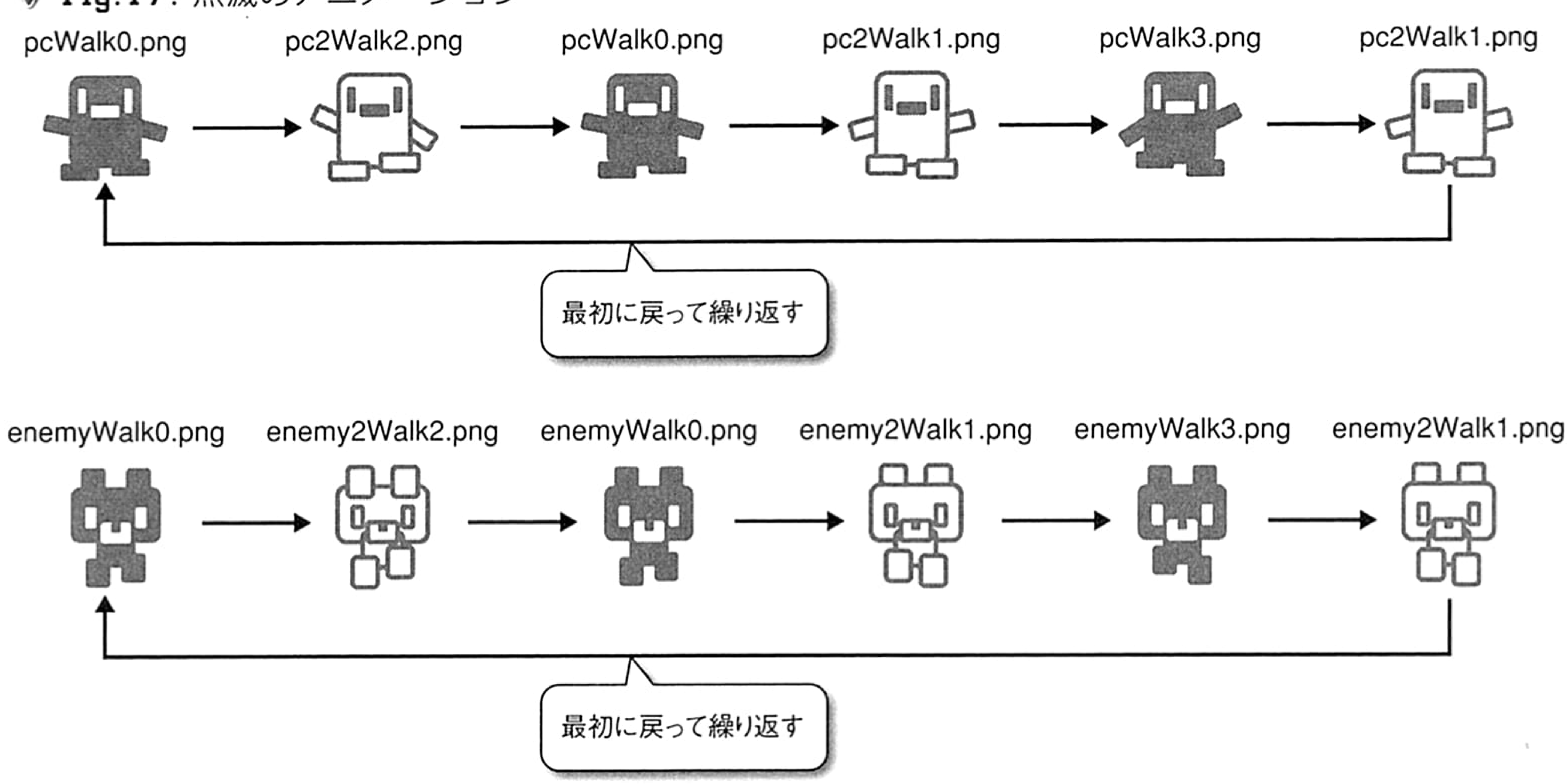
この動きはタイマーを使って実現します(Fig.16)。最初はタイマーを0に初期化しておき、ゲームの進行とともに(サンプルでは50ミリ秒ごとに)タイマーを1ずつ加算します。

Fig.16: パワーアップのタイマー



タイマーが点滅が始まる時間に達したら、自機と敵に点滅のアニメーションを設定します。点滅のアニメーションは、白色と黒色のキャラクターを交互に配置して作りました(Fig.17)。

Fig.17: 点滅のアニメーション

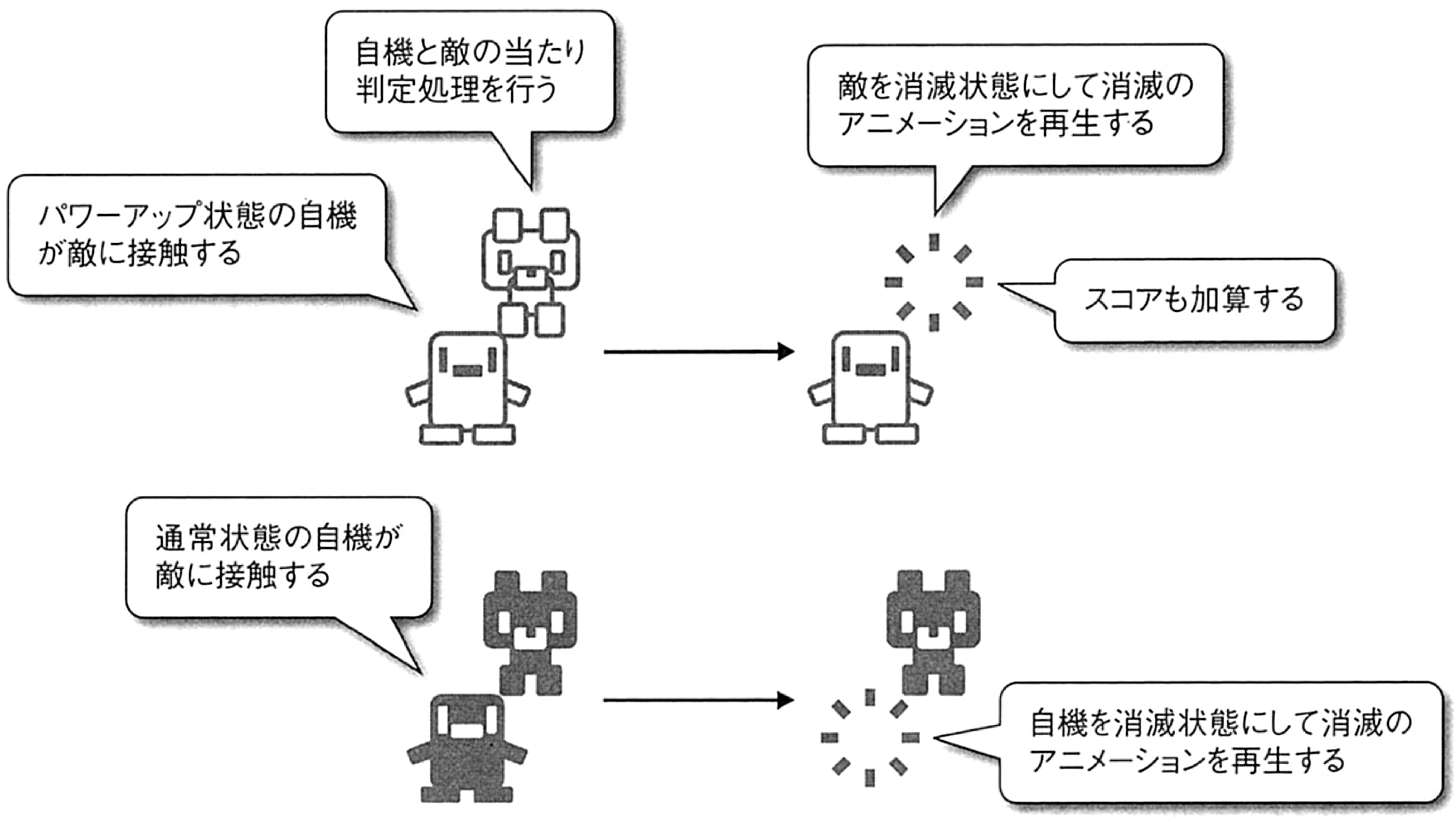


タイマーがパワーアップが終わる時間に達したら、自機と敵に通常時のアニメーションを設定します。パワーアップ中に自機に倒された敵がいたら、再び出現させます。

パワーアップ中に敵を倒す

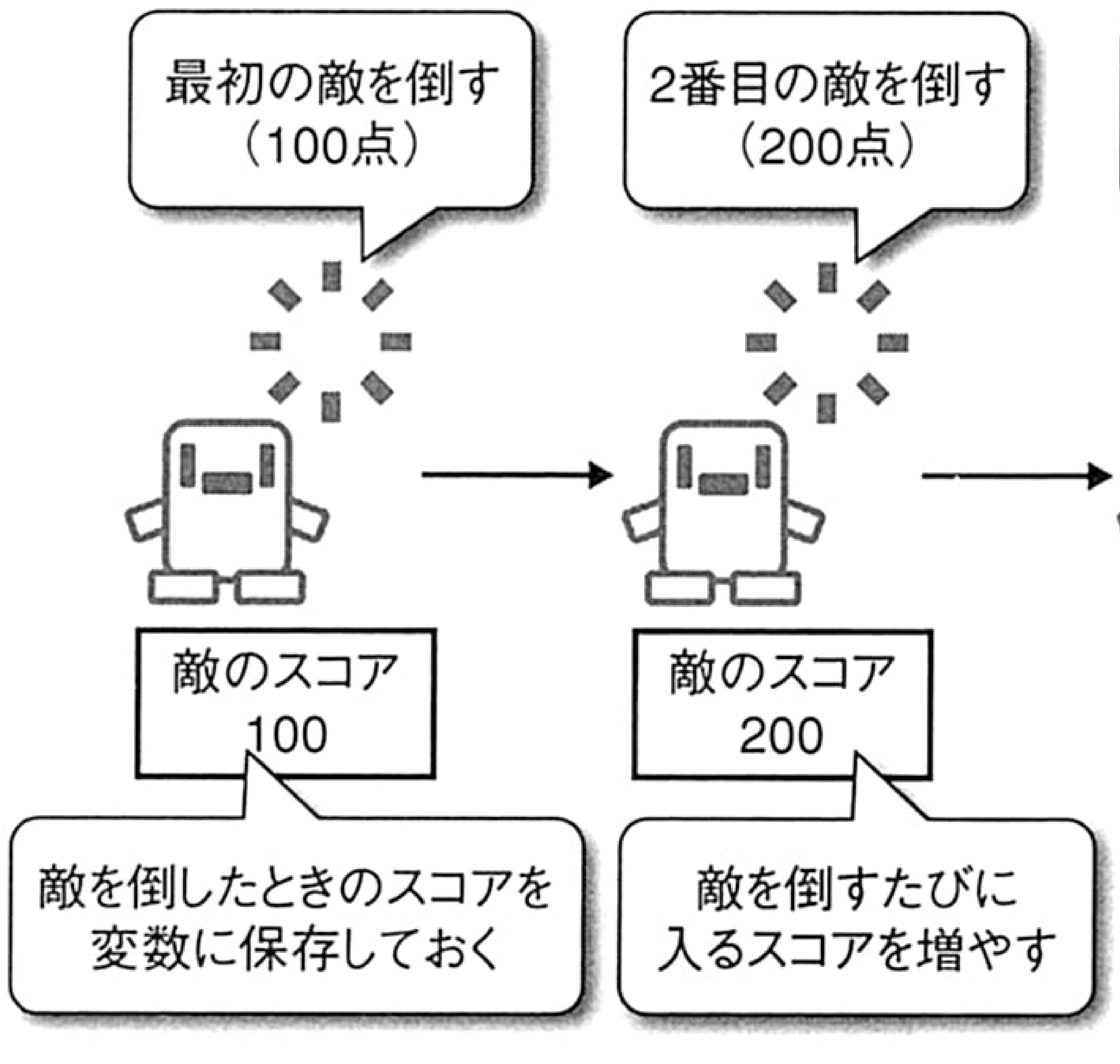
自機がパワーアップ中に敵と接触すると、敵を倒すことができます。敵と接触したかどうかは、自機と敵の当たり判定処理を行えば分かります(Fig.18)。パワーアップ状態の自機が敵に接触したら、敵を消滅させます。通常状態の自機が敵に接触したら、自機を消滅させます。

▼ Fig.18: 自機と敵の当たり判定処理



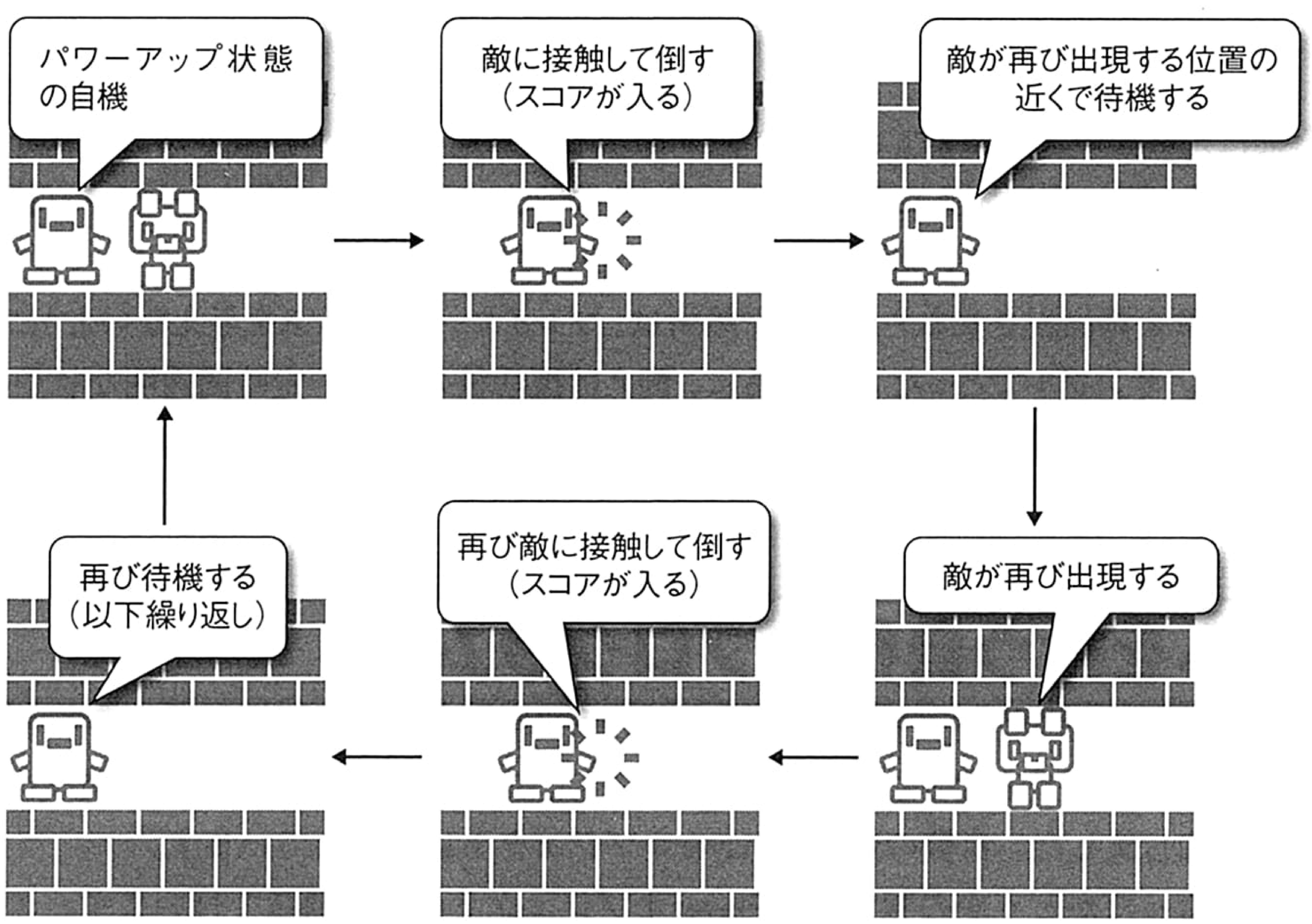
敵を消滅させるとスコアが入ります。連続して敵を倒したときに高いスコアが得られるようにするには、敵を倒したときに入るスコアを変数で保存しておき、敵を倒すたびに変数の値を増やします(Fig.19)。サンプルでは敵を倒すたびに、100･200･400･800といった具合に、入るスコアを倍にしています。

▼ **Fig.19**: 連続して敵を倒したときのスコア



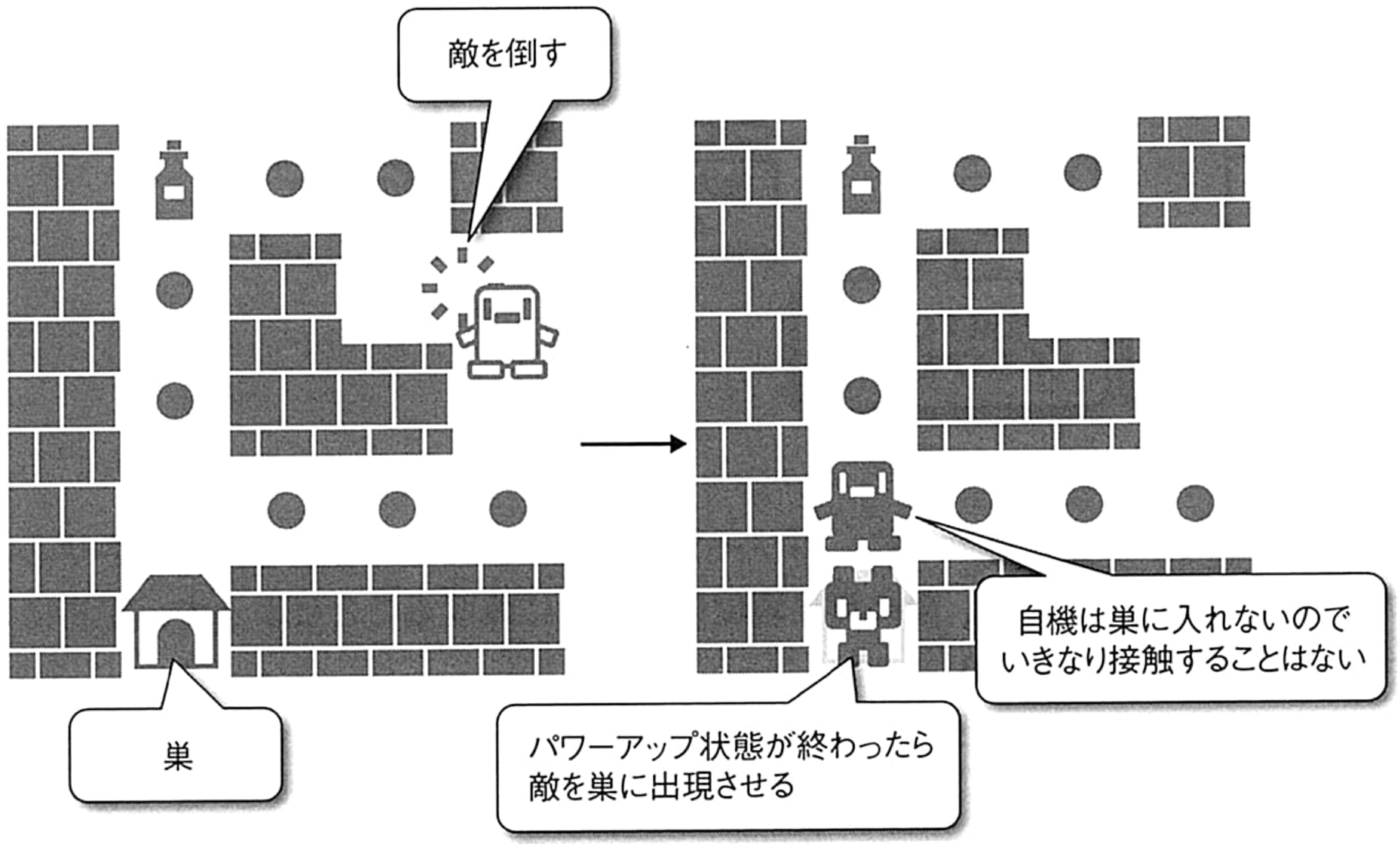
倒された敵はすぐには復活させずに、パワーアップ状態が終わってから再出現させることにしました。すぐに復活させると、敵の再出現位置の近くに陣取ることによって、安易にスコアが稼げてしまうからです（Fig.20）。

▼ **Fig.20**: 安易なスコア稼ぎ



パワーアップ状態が終わったら、フィールドの四隅にある巣から敵を再出現させます(Fig.21)。わざわざ巣を用意しているのは、何もない空間に敵が突然復活すると、そこを通った自機が不意に接触してしまうことがあるからです。

▼ **Fig.21**: 敵の再出現



自機を巣に入れないようにしておけば、敵が出現した瞬間に自機が接触することはなくなります。また、敵が巣に出現することは分かっているので、プレイヤーは「パワーアップ状態が終わったときには巣に近づかないようにしよう」と心がけることができます。

このように、敵を再出現させるタイミングや場所は、プレイヤーにストレスを与えないように決める必要があります。サンプルでとった方法以外に、例えば復活後の敵を一定時間透明な状態にして、自機が接触しても何も影響がない(自機も敵も倒されない)ようにする方法などもよいでしょう。

アイテムを拾う自機のプログラム

List1は自機のプログラムです。第7章のサンプルに、アイテムを拾う処理や、パワーアップする処理を追加しました。自機の処理はPlayerオブジェクトにまとめています。

アイテムを拾ったかどうかの判定は、Playerオブジェクトのalive関数で行います。エサアイテムについては、

```
if (field.getCell(cx, cy)==CELL_ITEM) {
```

のようなif文で判定します。(cx, cy)は自機のセル座標です。自機がいるセルを調べて、エサアイテムのセル(CELL_ITEM)ならば、エサアイテムを拾ったと判断します。getCellは指定した座標のセルを取得する関数です(p.184)。

エサアイテムを拾ったら、

```
field.setCell(cx, cy, CELL_VOID);
```

のようにセルを空間(CELL_VOID)にして、画面上からエサアイテムを消します。setCellは指定した座標のセルを設定する関数です(p.184)。

エサアイテムを拾ったらスコアが入るので、

```
game.addScore(SCORE_ITEM);
```

のようにスコアを加算します。また、

```
game.itemCount--;
```

のように、残ったエサアイテムの数を減らします。そして、

```
if (game.itemCount==0) {
    initStage();
}
```

のように、全てのエサアイテムを拾ったらステージを初期化して、新しいステージを開始します。

パワーアップアイテムを拾ったかどうかは、

```
if (field.getCell(cx, cy)==CELL_POWERUP) {
```

のように、エサアイテムと同じ要領で判定します。自機がパワーアップ状態でなければ、パワーアップアイテムを拾うので、

```
field.setCell(cx, cy, CELL_VOID);
```

のようにパワーアップアイテムを消します。そして、

```
obj.powerupTime=0;
obj.powerupScore=SCORE_ENEMY;
```

のように、タイマーや敵のスコアを初期化します。さらに、

```
obj.sprite.initAnim(playerPowerupAnim);
obj.sprite.setState(STATE_PLAYER_POWERUP);
```

のように、自機にパワーアップ状態のアニメーションを設定して、パワーアップ状態にします。

敵については、

```
for (i=0; i<enemy.length; i++) {
    spr=enemy[i].sprite;
    if (spr.getState()==STATE_ENEMY_ALIVE) {
        spr.initAnim(enemyEscapeAnim);
    }
}
```

のように、パワーアップ状態のときのアニメーション(逃亡のアニメーション)を設定します。

パワーアップ状態の自機が敵に接触したら、敵を倒します。この処理はonHitSprite関数で行います。onHitSpriteは他のスプライトに接触したときに呼び出される関数です(p.95)。

自機がパワーアップ状態のときに、通常状態の敵に接触したら、

```
spr.initAnim(vanishAnim);
spr.setState(STATE_ENEMY_DEAD);
```

のように、敵に消滅のアニメーションを設定して、消滅状態にします。また、

```
game.addScore(obj.powerupScore);
```

のようにスコアを加算し、

```
obj.powerupScore*=2;
```

のように、敵を倒したときのスコアを倍にします。最初に敵を倒すと100点入りますが、その

後パワーアップ状態が終わる前に別の敵を倒すと200点入ります。さらに連続して敵を倒すと400点·800点といった具合に、敵一体あたりのスコアが増えていきます。

パワーアップ状態は一定時間が経過すると終わります。パワーアップ状態のときには、

```
obj.powerupTime++;
```

のようにタイマーを更新し、一定時間が経過したかどうかを調べます。

点滅が始まる時間に達したかどうかは、

```
if (obj.powerupTime==PLAYER_BLINK_TIME) {
```

のようなif文で判定します。時間に達したら、

```
obj.sprite.initAnim(playerBlinkAnim);
```

のように、自機に点滅のアニメーションを設定します。敵についても同様に、点滅のアニメーションを再生させます。

パワーアップが終わる時間に達したかどうかは、

```
if (obj.powerupTime==PLAYER_POWERUP_TIME) {
```

のように判定します。時間に達したら、

```
obj.sprite.initAnim(playerWalkAnim);
obj.sprite.setState(STATE_PLAYER_ALIVE);
```

のように、自機に通常のアニメーションを設定し、通常状態に戻します。敵についても、通常のアニメーションを設定します。

List1: アイテムを拾う自機(mazeItem.js)

```
// グリッド(座標を補正するための定数)
var GRID=40;

// 自機のスピード
var PLAYER_SPEED=8/GRID;
```

```
// 点滅を始める時間・パワーアップが終わる時間
var PLAYER_BLINK_TIME=50;
var PLAYER_POWERUP_TIME=60;

// 自機の状態(通常・消滅・パワーアップ)
var STATE_PLAYER_ALIVE=0;
var STATE_PLAYER_DEAD=1;
var STATE_PLAYER_POWERUP=2;

// スコア(エサアイテムを拾ったとき・敵を倒したとき)
var SCORE_ITEM=10;
var SCORE_ENEMY=100;

// セルの種類(空間・壁・エサアイテム・パワーアップアイテム・巣)
var CELL_VOID=0;
var CELL_WALL=1;
var CELL_ITEM=2;
var CELL_POWERUP=3;
var CELL_HOUSE=4;

// 指定した座標のセルが通れないとき(壁または巣のとき)trueを返す
function isWall(x, y) {
  var cell=field.getCell(x, y);
  return cell==CELL_WALL || cell==CELL_HOUSE;
}

// 自機
function Player(x, y) {
  var obj=this;

  // スプライトの作成
  obj.sprite=game.createSprite();

  // アニメーション・当たり判定・グリッドの設定
  obj.sprite.initAnim(playerWalkAnim);
  obj.sprite.initHit(playerHit);
  obj.sprite.setGrid(GRID);

  // 座標とスクロール座標の設定
  obj.sprite.setXY(x, y);
  game.setViewXY(x-XMAX/2, y-YMAX/2);

  // 状態・速度・セル座標の設定
  obj.sprite.setState(STATE_PLAYER_ALIVE);
  obj.vx=0;
  obj.vy=0;
  obj.cx=x;
```

```
obj.cy=y;

// 更新処理
obj.sprite.onUpdate=function() {

  // 状態に応じて分岐する
  switch (obj.sprite.getState()) {

    // 通常状態とパワーアップ状態
    case STATE_PLAYER_ALIVE:
    case STATE_PLAYER_POWERUP:
      obj.alive();
      break;

    // 消滅状態
    case STATE_PLAYER_DEAD:
      obj.dead();
      break;
  }
}

// 通常状態とパワーアップ状態
obj.alive=function() {

  // 座標·速度·セル座標の取得
  var x=obj.sprite.getX(), y=obj.sprite.getY();
  var vx=obj.vx, vy=obj.vy;
  var cx=obj.cx, cy=obj.cy;
  var spr;

  // 自機の座標が整数のとき(座標とセル座標が一致しているとき)
  if (x==cx && y==cy) {

    // エサアイテムに接触したとき
    if (field.getCell(cx, cy)==CELL_ITEM) {

      // エサアイテムを消す(空間のセルにする)
      field.setCell(cx, cy, CELL_VOID);

      // スコアを加算する
      game.addScore(SCORE_ITEM);

      // 残りのエサアイテム数を減らす
      game.itemCount--;

      // 全てのアイテムを拾ったらステージを初期化する
      if (game.itemCount==0) {
```

```
      initStage();
    }
  }

  // パワーアップアイテムに接触したとき
  if (field.getCell(cx, cy)==CELL_POWERUP) {

    // 自機がパワーアップしていないとき
    if (obj.sprite.getState()!=STATE_PLAYER_POWERUP) {

      // パワーアップアイテムを消す(空間のセルにする)
      field.setCell(cx, cy, CELL_VOID);

      // タイマーを初期化する
      obj.powerupTime=0;

      // 敵のスコアを初期化する
      obj.powerupScore=SCORE_ENEMY;

      // パワーアップ時のアニメーションを設定する
      obj.sprite.initAnim(playerPowerupAnim);

      // パワーアップ状態にする
      obj.sprite.setState(STATE_PLAYER_POWERUP);

      // 敵もパワーアップ時のアニメーションにする
      for (i=0; i<enemy.length; i++) {
        spr=enemy[i].sprite;
        if (spr.getState()==STATE_ENEMY_ALIVE) {
          spr.initAnim(enemyEscapeAnim);
        }
      }
    }
  }

  // カーソルキーの入力に応じて上下左右に移動する
  vx=vy=0;
  if (game.isKey(KEY_LEFT)) vx=-1; else
  if (game.isKey(KEY_RIGHT)) vx=1; else
  if (game.isKey(KEY_UP)) vy=-1; else
  if (game.isKey(KEY_DOWN)) vy=1;

  // 移動先が壁のときは移動しない
  if (isWall(cx+vx, cy+vy)) {
    vx=vy=0;
  }
```

```
    // セル座標の更新
    cx+=vx;
    cy+=vy;
  }

  // 座標の更新
  x+=vx*PLAYER_SPEED;
  y+=vy*PLAYER_SPEED;
  obj.sprite.setXY(x, y);

  // 速度とセル座標の更新
  obj.vx=vx;
  obj.vy=vy;
  obj.cx=cx;
  obj.cy=cy;

  // 移動中はアニメーションを再生する
  if (vx!=0 || vy!=0) {
    obj.sprite.stepAnim();
  }

  // 自機が画面中央にくるようにスクロール座標を更新する
  game.setViewXY(x-XMAX/2, y-YMAX/2);

  // 敵と接触したときの処理
  obj.sprite.onHitSprite=function(spr) {

    // 通常状態の敵に接触したとき
    if (spr.getState()==STATE_ENEMY_ALIVE) {

      // 自機がパワーアップ状態のとき
      if (obj.sprite.getState()==STATE_PLAYER_POWERUP) {

        // 敵に消滅のアニメーションを設定する
        spr.initAnim(vanishAnim);

        // 敵を消滅状態にする
        spr.setState(STATE_ENEMY_DEAD);

        // スコアを加算する
        game.addScore(obj.powerupScore);

        // 敵を倒したときのスコアを倍にする
        obj.powerupScore*=2;
      } else

      // 自機が通常状態のとき
```

```
      {
        // 自機に消滅のアニメーションを設定する
        obj.sprite.initAnim(vanishAnim);

        // 自機を消滅状態にする
        obj.sprite.setState(STATE_PLAYER_DEAD);
      }
    }
  }

  // 敵との当たり判定処理
  game.testHitSprite(obj.sprite);

  // パワーアップ状態の処理
  if (obj.sprite.getState()==STATE_PLAYER_POWERUP) {

    // タイマーを更新する
    obj.powerupTime++;

    // 点滅が始まる時間に達したとき
    if (obj.powerupTime==PLAYER_BLINK_TIME) {

      // 自機に点滅のアニメーションを設定する
      obj.sprite.initAnim(playerBlinkAnim);

      // 敵に点滅のアニメーションを設定する
      for (i=0; i<enemy.length; i++) {
        spr=enemy[i].sprite;
        if (spr.getState()==STATE_ENEMY_ALIVE) {
          spr.initAnim(enemyBlinkAnim);
        }
      }
    } else

    // パワーアップが終わる時間に達したとき
    if (obj.powerupTime==PLAYER_POWERUP_TIME) {

      // 自機に通常のアニメーションを設定する
      obj.sprite.initAnim(playerWalkAnim);

      // 自機を通常状態にする
      obj.sprite.setState(STATE_PLAYER_ALIVE);

      // 敵に通常のアニメーションを設定する
      for (i=0; i<enemy.length; i++) {
        spr=enemy[i].sprite;
        if (spr.getState()==STATE_ENEMY_ALIVE) {
```

```
            spr.initAnim(enemyWalkAnim);
          }
        }
      }
    }
  }

  // 消滅状態
  obj.dead=function() {

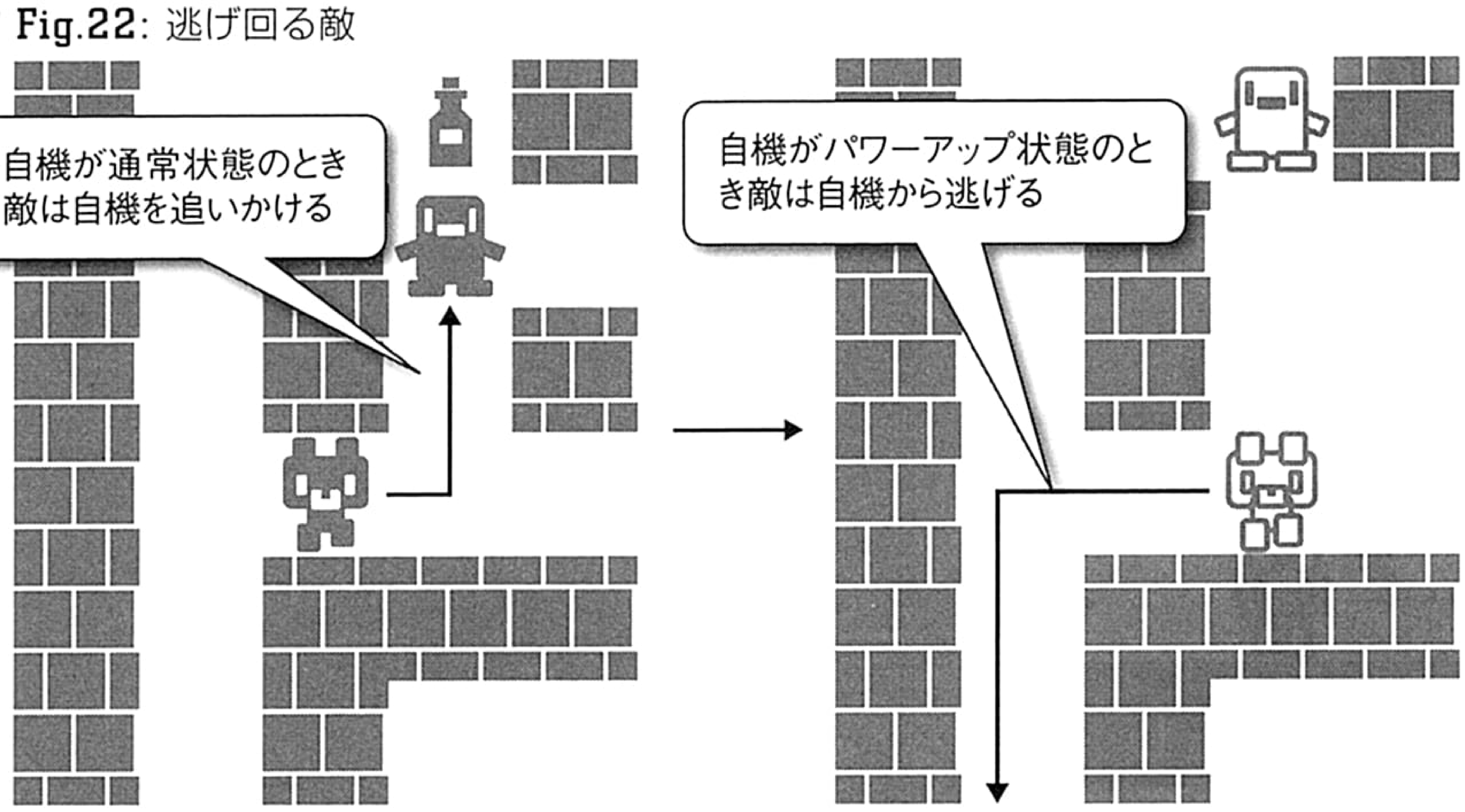
    // アニメーションを再生する
    obj.sprite.stepAnim();

    // アニメーションが終わったらゲームを初期化する
    if (obj.sprite.isAnimOver()) {
      initGame();
    }
  }
}
```

自機から逃げる敵

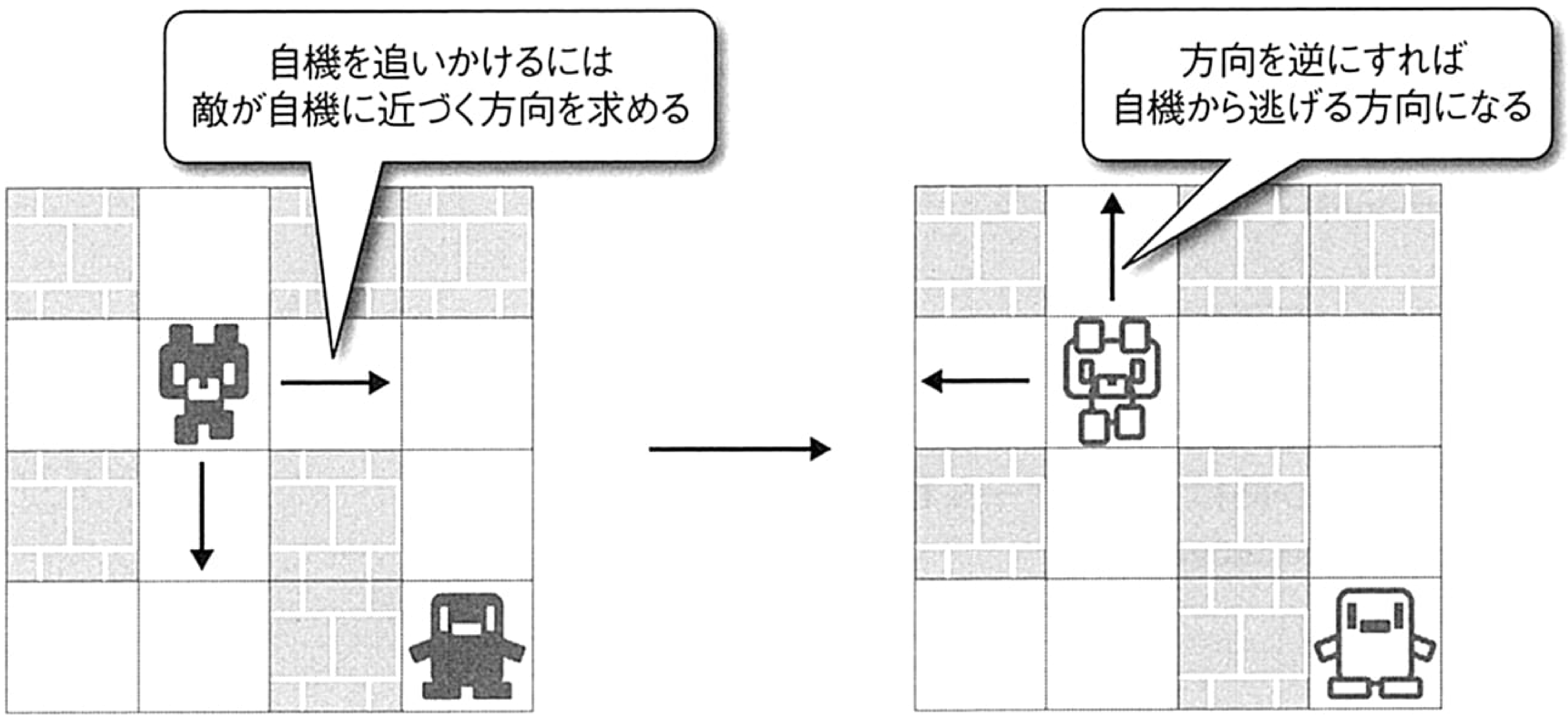
敵はフィールドを動き回り、自機を追いかけてきます。ここまでは第7章のサンプルと同じなのですが、自機がパワーアップしたときには、逆に自機につかまらないように逃げ回ります(Fig.22)。

▼ **Fig.22**: 逃げ回る敵



自機から逃げる敵を作るのは、実はとても簡単です。自機を追いかける敵の移動方向を逆にすればよいのです(Fig.23)。

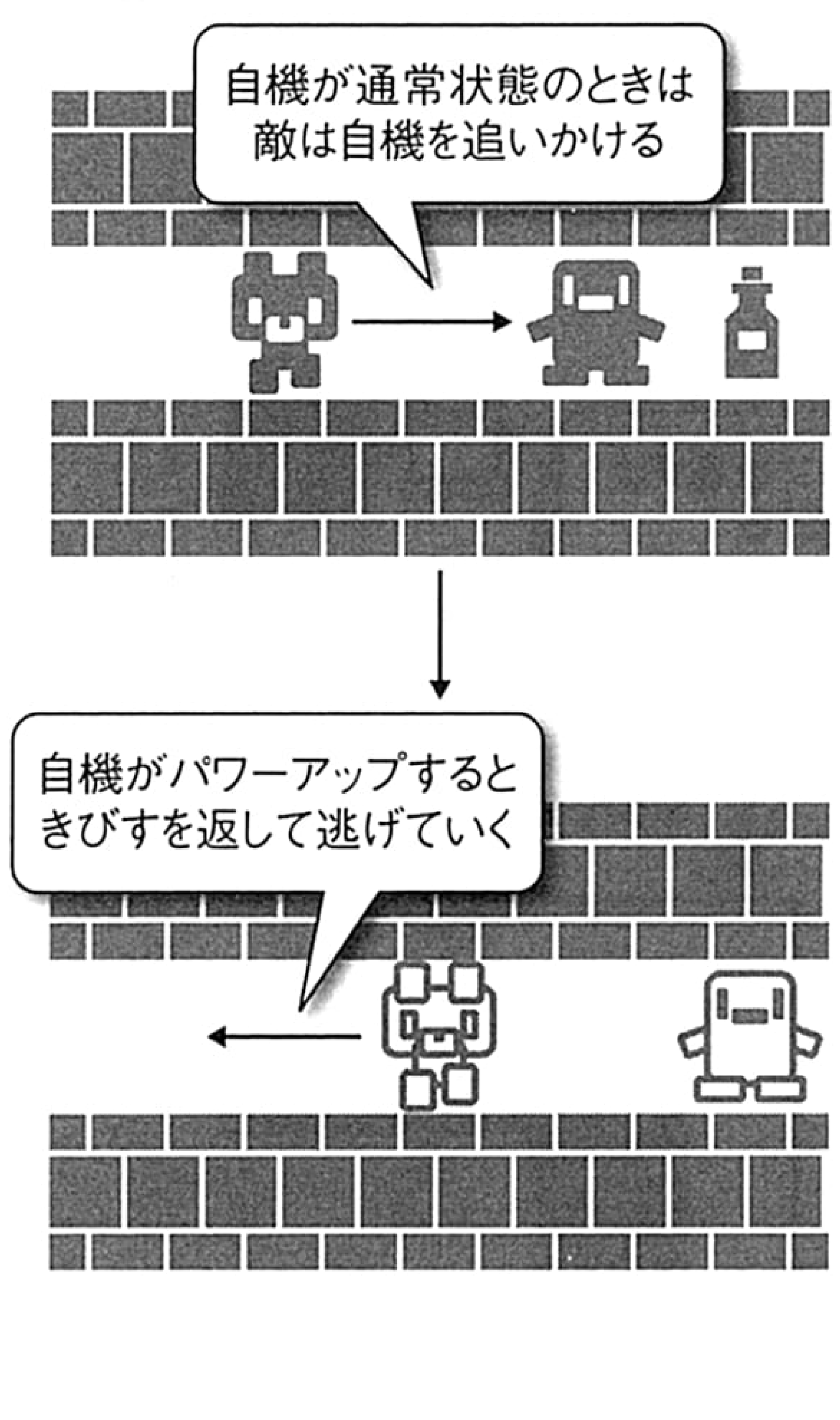
▼ **Fig.23**: 敵の移動方向を逆にする



自機を追いかけるというのは、自機に近づく方向に移動するということです。移動方向を逆にすれば、自機から遠ざかることになります。これはつまり、自機から逃げ回るということです。

自機がパワーアップした瞬間には、敵が驚いてきびすを返すと面白そうです(Fig.24)。通常は敵の動きが自然になるように、敵が後戻りをしないようにしていますが、パワーアップした瞬間だけは後戻りを許すことにします。

▼ **Fig.24**: 驚いてきびすを返す敵



具体的には、前回の移動方向とは関係なく、敵が自機から遠ざかる方向へすぐに移動できるようにします。これでパワーアップアイテムを取った瞬間に敵がきびすを返すようになり、追うものと追われるものの立場が入れ替わったことがはっきりして、ゲームの雰囲気が盛り上がります。

自機から逃げる敵のプログラム

List2は自機から逃げる敵のプログラムです。常に逃げ回るわけではなく、自機が通常状態のときには追いかけて、パワーアップ状態のときには逃げます。

敵を動かすのはEnemyオブジェクトのalive関数です。まず、

```
vx=vy=0;
if (px<cx) vx=-1; else if (px>cx) vx=1;
if (py<cy) vy=-1; else if (py>cy) vy=1;
```

のように、自機のセル座標(px, py)と敵のセル座標(cx, cy)を使って、自機に近づく方向(vx, vy)を求めます。ここまでは第7章のサンプルと同じです(p.212)。

次に、自機がパワーアップ状態かどうかを調べます。パワーアップ状態ならば、

```
vx=-vx;
vy=-vy;
```

として、移動方向の符号を反転させます。これで自機から遠ざかる方向になります。

自機がパワーアップした瞬間には、敵がきびすを返せるようにします。これは、

```
if (obj.playerState==STATE_PLAYER_ALIVE) {
    ovx=ovy=0;
}
```

のように行います。(ovx, ovy)は前回の移動方向です。通常はこの(ovx, ovy)を使って、敵が後戻りしないようにしています。自機がパワーアップした瞬間だけはこれを無効にして、前回の移動方向とは関係なく、敵が自機から遠ざかる方向へすぐに移動できるようにします。

敵は自機に倒されると消滅します。消滅状態の処理を行うdead関数では、

```
if (!obj.sprite.isAnimOver()) {
    obj.sprite.stepAnim();
}
```

のように、消滅のアニメーションが終わるまでは、アニメーションを再生します。アニメーションが終わったら、画面から消えたまま待機します。

自機のパワーアップ状態が終わったら、

```
if (player.sprite.getState()!=STATE_PLAYER_POWERUP) {
    obj.init();
}
```

のように、敵を再出現させます。敵はフィールドの四隅にある巣から復活します。

List2: 自機から逃げる敵(mazeItem.js)

```
// 敵のスピード
var ENEMY_SPEED=5/GRID;

// 敵の状態(通常・消滅)
var STATE_ENEMY_ALIVE=3;
var STATE_ENEMY_DEAD=4;

// 敵
function Enemy(x, y) {
  var obj=this;

  // スプライトの作成
  obj.sprite=game.createSprite();

  // 当たり判定・グリッド・初期座標の設定
  obj.sprite.initHit(enemyHit);
  obj.sprite.setGrid(GRID);
  obj.initX=x;
  obj.initY=y;

  // 敵の初期化
  obj.init=function() {

    // アニメーション・座標・状態・速度の設定
    obj.sprite.initAnim(enemyWalkAnim);
    obj.sprite.setXY(obj.initX, obj.initY);
    obj.sprite.setState(STATE_ENEMY_ALIVE);
    obj.vx=0;
    obj.vy=0;

    // 前回の自機の状態
    obj.playerState=STATE_PLAYER_ALIVE;
  }
  obj.init();
```

```
// 更新処理
obj.sprite.onUpdate=function() {

  // 状態に応じて分岐する
  switch (obj.sprite.getState()) {
    case STATE_ENEMY_ALIVE: obj.alive(); break;
    case STATE_ENEMY_DEAD: obj.dead(); break;
  }
}

// 通常状態
obj.alive=function() {

  // 座標·速度·古い速度の取得
  var x=obj.sprite.getX(), y=obj.sprite.getY();
  var vx=obj.vx, vy=obj.vy;
  var ovx=obj.vx, ovy=obj.vy;

  // 敵のセル座標·自機のセル座標の取得
  var cx=Math.floor(x), cy=Math.floor(y);
  var px=player.cx, py=player.cy;

  // 自機の状態の取得
  var ps=player.sprite.getState();

  // 敵の座標が整数のとき(座標とセル座標が一致しているとき)
  if (x==cx && y==cy) {

    // 自機に近づく方向を求める
    vx=vy=0;
    if (px<cx) vx=-1; else if (px>cx) vx=1;
    if (py<cy) vy=-1; else if (py>cy) vy=1;

    // 自機がパワーアップ状態ならば、
    // 近づく方向の符号を反転させて、
    // 自機から遠ざかる方向にする
    if (ps==STATE_PLAYER_POWERUP) {

      // 方向を反転させる
      vx=-vx;
      vy=-vy;

      // 自機がパワーアップ状態に変わった瞬間ならば、
      // 古い速度(前回の移動方向)を初期化して、
      // 今回だけ後戻りできるようにする
      if (obj.playerState==STATE_PLAYER_ALIVE) {
        ovx=ovy=0;
```

```
        }
      }

      // 自機の状態を保存する
      obj.playerState=ps;

      // 求めた方向に壁があるときや、
      // 後戻りするときには移動をやめる
      if (isWall(cx+vx, cy) || vx==-ovx) vx=0;
      if (isWall(cx, cy+vy) || vy==-ovy) vy=0;

      // X方向·Y方向のどちらにも移動できるときには、
      // 移動方向をランダムに選ぶ
      if (vx!=0 && vy!=0) {
        if (randInt(2)==0) vx=0; else vy=0;
      } else

      // X方向·Y方向のどちらにも移動できないときには、
      // 壁がないいずれかの方向に移動する
      if (vx==0 && vy==0) {

        // 古い速度(ovx, ovy)を調べて、
        // 来た方向に後戻りしないようにする
        if (!isWall(cx-1, cy) && ovx!=1) vx=-1; else
        if (!isWall(cx+1, cy) && ovx!=-1) vx=1; else
        if (!isWall(cx, cy-1) && ovy!=1) vy=-1; else
        if (!isWall(cx, cy+1) && ovy!=-1) vy=1;
      }
    }

    // 座標と速度の更新
    obj.sprite.setXY(x+vx*ENEMY_SPEED, y+vy*ENEMY_SPEED);
    obj.vx=vx;
    obj.vy=vy;

    // アニメーションの再生
    obj.sprite.stepAnim();
  }

  // 消滅状態
  obj.dead=function() {

    // アニメーションが終わらない限り、
    // 消滅のアニメーションを再生する
    if (!obj.sprite.isAnimOver()) {
      obj.sprite.stepAnim();
    } else
```

```
      // アニメーションが終わっており、
      // 自機が通常状態に戻ったら、
      // 敵を初期化して再出現させる
      if (player.sprite.getState()!=STATE_PLAYER_POWERUP) {
        obj.init();
      }
    }
  }
```

アイテムを拾うゲームの本体

List3はアイテムを拾うゲームの本体です。他のサンプルと同様に、ゲームの初期化処理とメインルーチンを定義します。

初期化処理は、ステージの初期化を行うinitStage関数と、ゲームの初期化を行うinitGame関数に分けました。このサンプルでは全てのエサアイテムを拾うとステージクリアとなり、ステージの再初期化が必要になるからです。ゲームの初期化では、

```
function initGame() {
    game.setScore(0);
    initStage();
}
```

のように、スコアを0にした上で、ステージを初期化します。

ステージを初期化するときには、

```
game.itemCount=0;
for (y=0; y<field.getYCount(); y++) {
    for (x=0; x<field.getXCount(); x++) {
        if (field.getCell(x, y)==CELL_ITEM) {
            game.itemCount++;
        }
    }
}
```

のように、フィールドにあるエサアイテムを数えます。for文を使って全てのセルを調べ、エサアイテムのセルを見つけたら、個数(itemCountプロパティ)に1を加えます。エサアイテムは数が多いので、このようにプログラムで数えるのが簡単で、間違いもありません。

▼ **List3**: アイテムを拾うゲームの本体(mazeItem.js)

```
// ゲーム本体・自機・敵・フィールドのオブジェクト
var game;
var player;
var enemy;
var field;

// 自機のアニメーション(通常・パワーアップ・点滅)
var playerWalkAnim;
var playerPowerupAnim;
var playerBlinkAnim;

// 敵のアニメーション(通常・逃亡・点滅)
var enemyWalkAnim;
var enemyEscapeAnim;
var enemyBlinkAnim;

// 消滅のアニメーション
var vanishAnim;

// 自機・敵の当たり判定
var playerHit;
var enemyHit;

// ステージの初期化
function initStage() {
  var x, y;

  // 画面の消去
  game.clear();

  // フィールドの作成
  field=game.createField();

  // セルの種類に対応する画像の一覧を配列で設定する
  field.initCellImage(new Array(
    "void.png", "wall.png", "item.png",
    "powerup.png", "house.png"
  ));

  // セルの初期化
  field.initCell(new Array(
    "0000000000000000000000000",
    "0000011100000000011100000",
    "0000014111111111114100000",
    "0000012222221222222100000",
    "0000012111123211112100000",
```

```
    "0000012222121212222100000",
    "0000012112121212112100000",
    "0000012122220222212100000",
    "0000013221211121223100000",
    "0000012122222222212100000",
    "0000012112121212112100000",
    "0000012222121212222100000",
    "0000012111123211112100000",
    "0000012222221222222100000",
    "0000014111111111114100000",
    "0000011100000000011100000",
    "0000000000000000000000000"
  ));

  // アイテムを数える
  game.itemCount=0;
  for (y=0; y<field.getYCount(); y++) {
    for (x=0; x<field.getXCount(); x++) {
      if (field.getCell(x, y)==CELL_ITEM) {
        game.itemCount++;
      }
    }
  }

  // 自機の作成
  player=new Player(12, 7);

  // 敵の作成
  enemy=new Array();
  enemy[0]=new Enemy(6, 2);
  enemy[1]=new Enemy(18, 2);
  enemy[2]=new Enemy(6, 14);
  enemy[3]=new Enemy(18, 14);
}

// ゲームの初期化
function initGame() {

  // スコアの初期化
  game.setScore(0);

  // ステージの初期化
  initStage();
}

// メインルーチン
function main() {
```

```
  // ゲーム本体の作成
  game=new Game();

  // アニメーションの作成
  playerWalkAnim=new Array(
    "pcWalk0.png", "pcWalk2.png", "pcWalk0.png",
    "pcWalk1.png", "pcWalk3.png", "pcWalk1.png"
  );
  playerPowerupAnim=new Array(
    "pc2Walk0.png", "pc2Walk2.png", "pc2Walk0.png",
    "pc2Walk1.png", "pc2Walk3.png", "pc2Walk1.png"
  );
  playerBlinkAnim=new Array(
    "pcWalk0.png", "pc2Walk2.png", "pcWalk0.png",
    "pc2Walk1.png", "pcWalk3.png", "pc2Walk1.png"
  );
  enemyWalkAnim=new Array(
    "enemyWalk0.png", "enemyWalk2.png", "enemyWalk0.png",
    "enemyWalk1.png", "enemyWalk3.png", "enemyWalk1.png"
  );
  enemyEscapeAnim=new Array(
    "enemy2Walk0.png", "enemy2Walk2.png", "enemy2Walk0.png",
    "enemy2Walk1.png", "enemy2Walk3.png", "enemy2Walk1.png"
  );
  enemyBlinkAnim=new Array(
    "enemyWalk0.png", "enemy2Walk2.png", "enemyWalk0.png",
    "enemy2Walk1.png", "enemyWalk3.png", "enemy2Walk1.png"
  );
  vanishAnim=new Array(
    "void.png", "vanish8.png", "vanish6.png",
    "vanish4.png", "vanish2.png", "vanish0.png",
    "vanish1.png", "vanish2.png", "vanish3.png",
    "vanish4.png", "vanish5.png", "vanish6.png",
    "vanish7.png", "vanish8.png", "void.png"
  );

  // 当たり判定の作成
  playerHit=new Array(0.25, 0.25, 0.75, 0.75);
  enemyHit=new Array(0.25, 0.25, 0.75, 0.75);

  // ゲームの初期化
  initGame();
}
```

Chapter 9

ゲームのエッセンス

色々なゲームのアクションを知る

オリジナリティのあるゲームを作るには、既存のゲームをよく研究することが欠かせません。全く新しいアイディアを考えついたと思っても、それはありふれたアクションかもしれません。典型的なパターンを知った上で、発想を転換することによってこそ、今までになかったような要素を考え出すことができます。

そこで本章では、ゲームで使われる典型的なアクションの中から、

- 荷物を押して動かす
- トランポリン
- 床を押し上げる

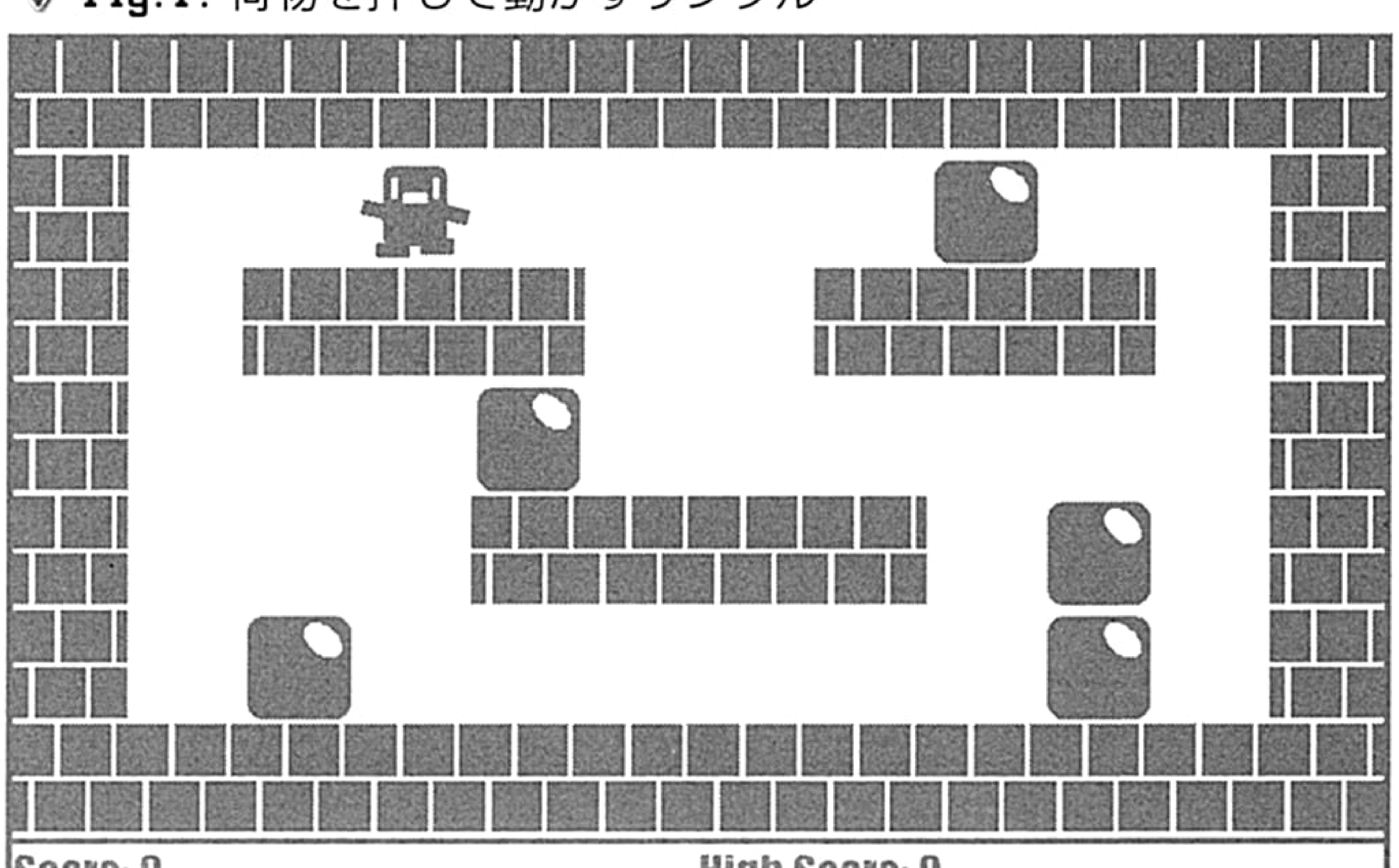
という3つのアクションを紹介します。こういった基本的な動きは色々なゲームで使えるので、知っておいて損はありません。また、動きをアレンジしてオリジナルのアクションを考案するのにも役立つでしょう。

荷物を押して動かす

最初は荷物を押して動かすアクションです。このアクションを採用した古典的なゲームは「倉庫番」(シンキングラビット 1982年)や「フラッピー」(デービーソフト 1983年)などです。そのほか、数多くのゲームが荷物を押して動かすアクションを採用しています。

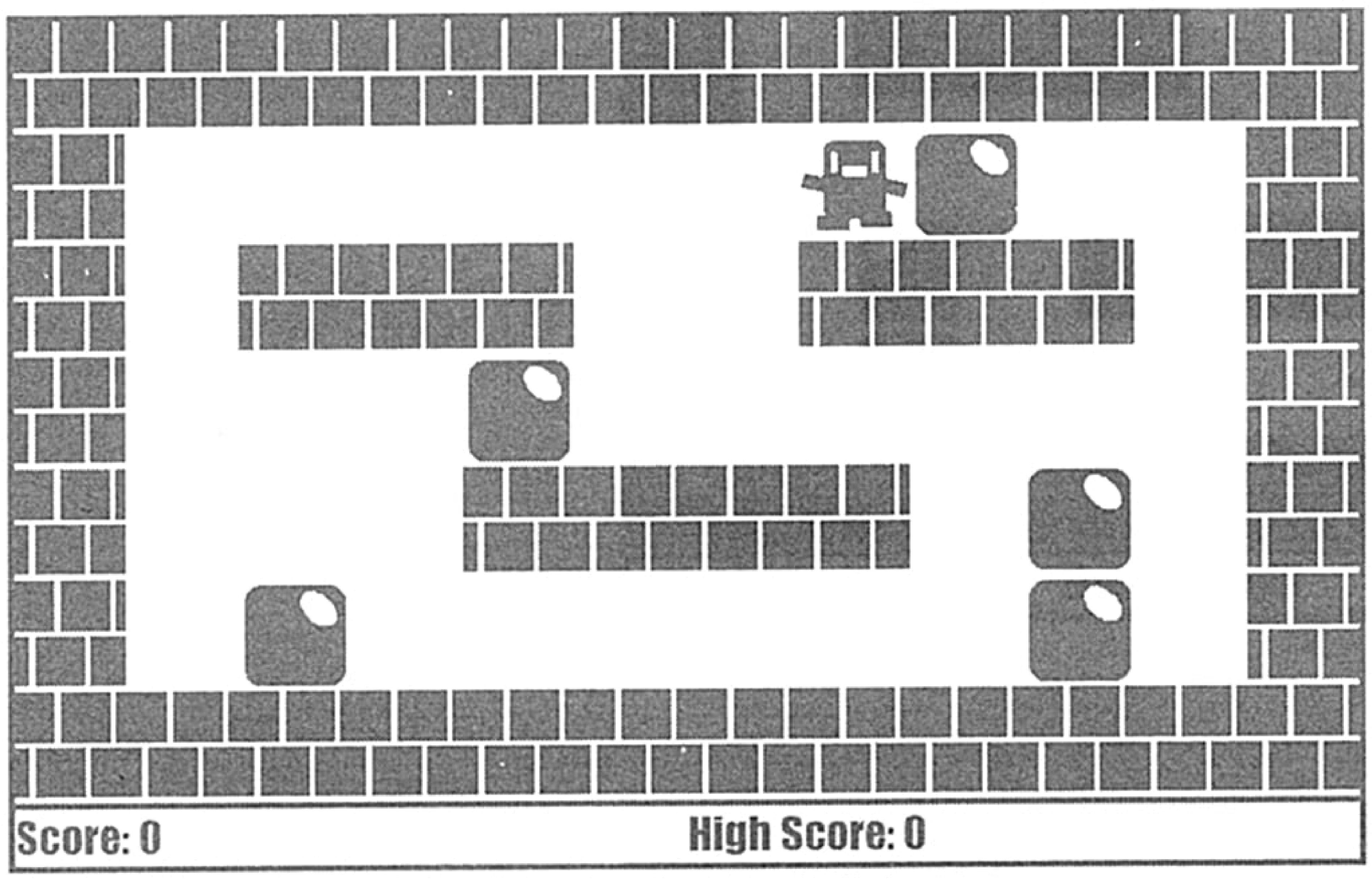
荷物を押して動かすサンプルを実行するには、pushLoad.htmlをブラウザで開きます(Fig.1)。自機は左上に表示されます。迷路の中には、いくつかの荷物が配置されています。

Fig.1: 荷物を押して動かすサンプル



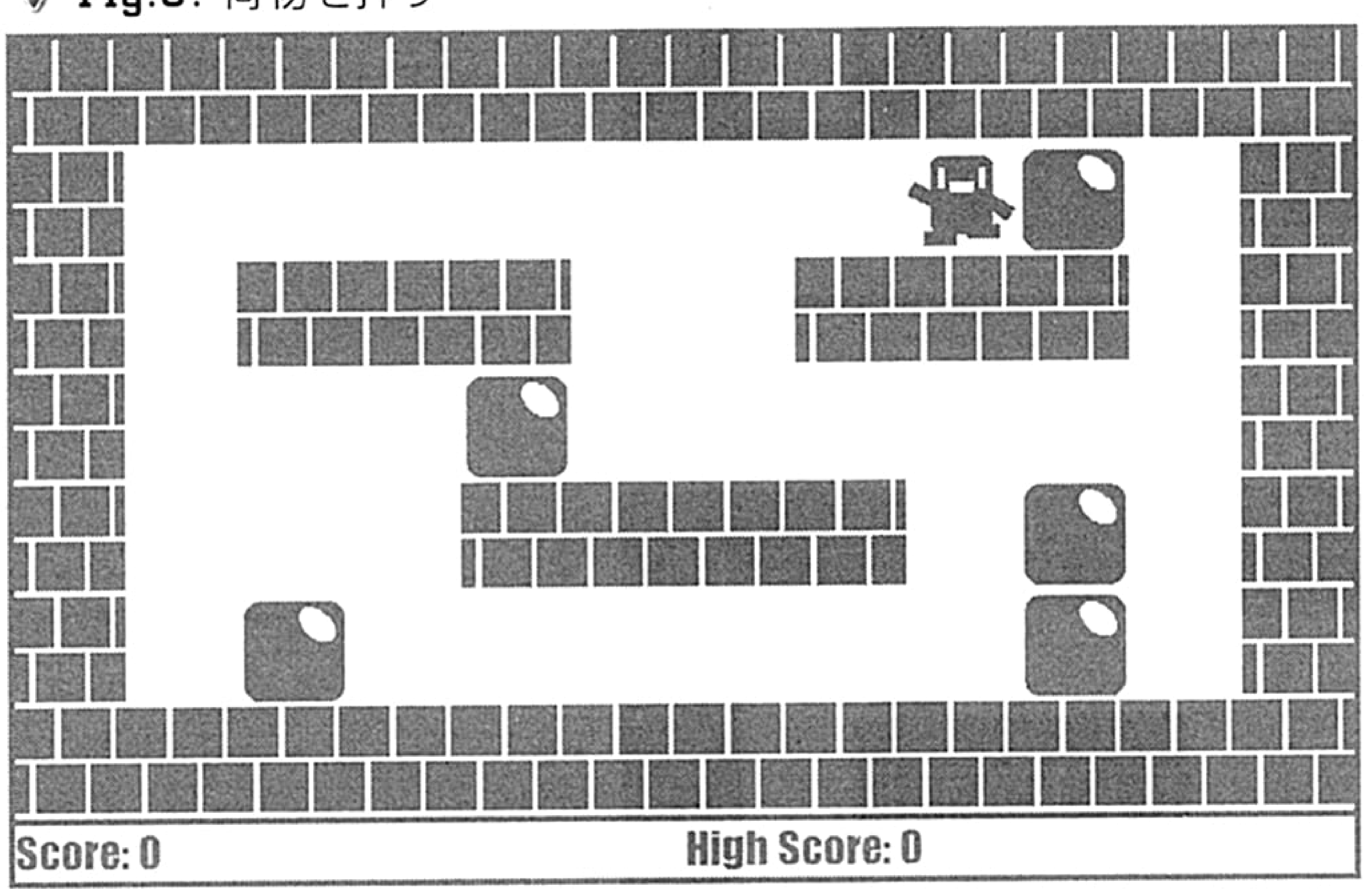
自機はカーソルキーで上下左右に動きます。自機を動かして、荷物に隣接させてみましょう(Fig.2)。

▼ Fig.2: 自機を荷物に隣接させる



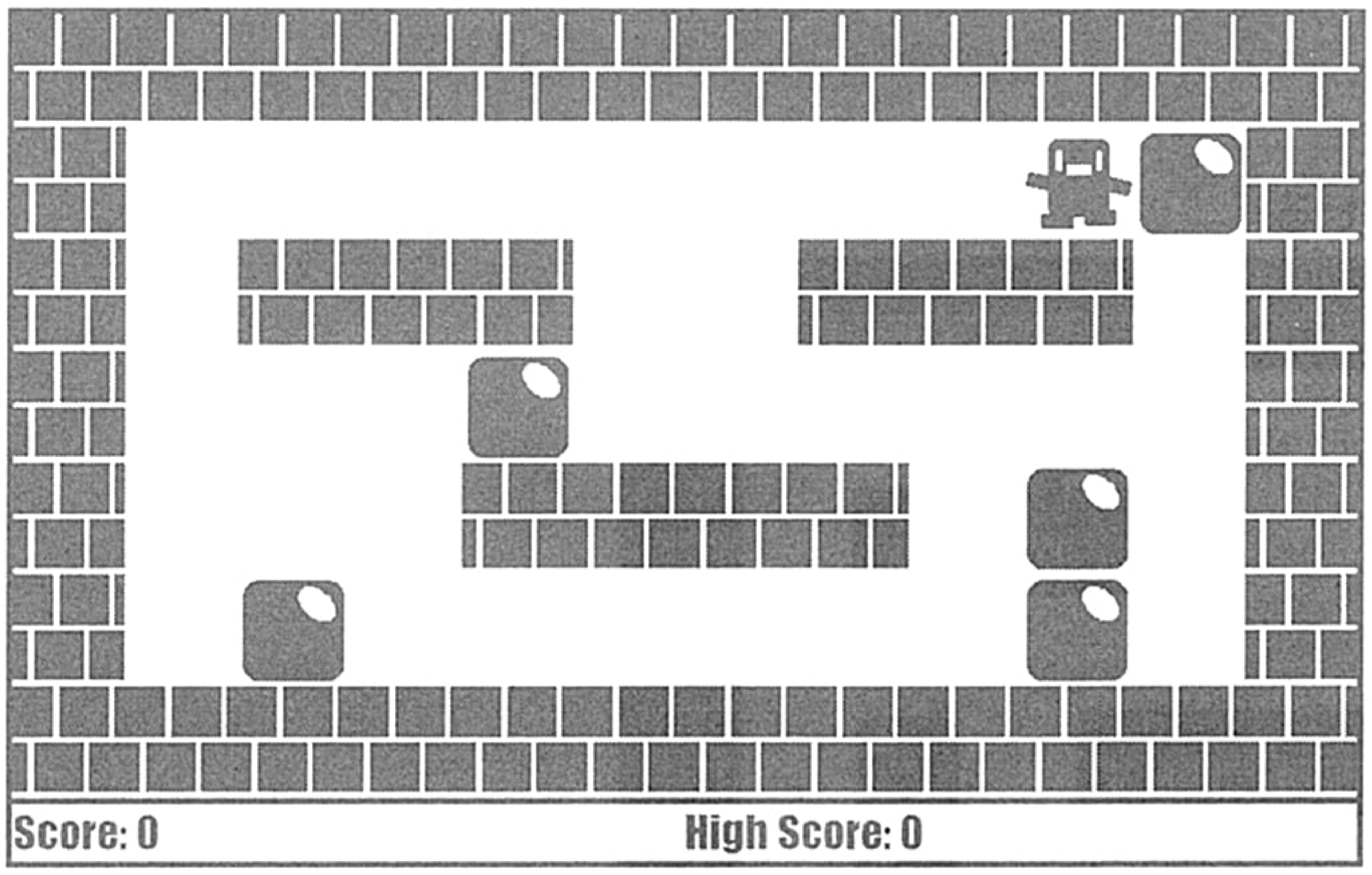
荷物に隣接した状態で、自機を荷物の方に動かすと、自機が荷物を押します(Fig.3)。荷物の前方に何もなければ、荷物は押した方向に動きます。前方に壁や他の荷物があるときには、荷物は動きません。

▼ Fig.3: 荷物を押す

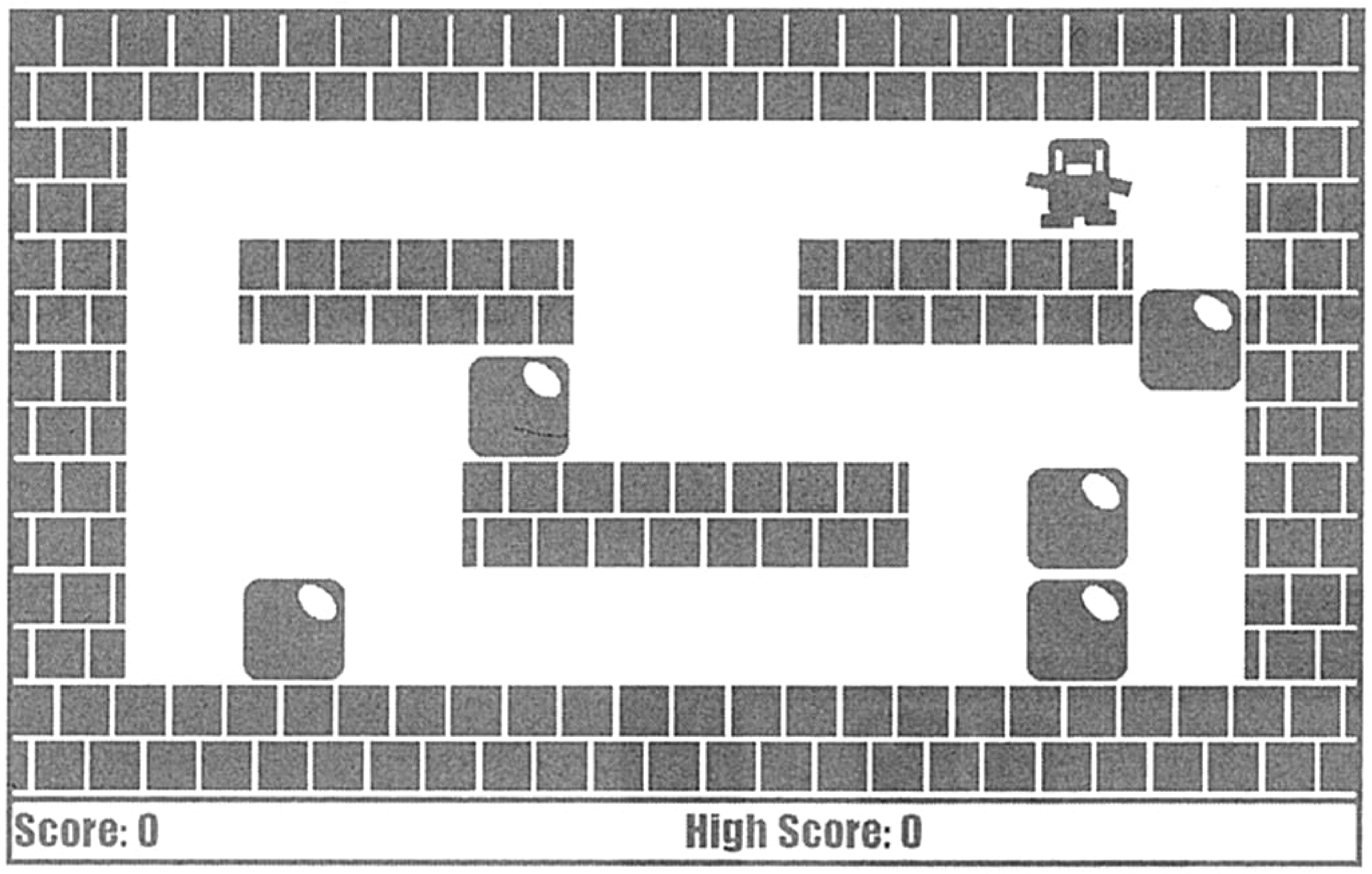


荷物を押した結果、下に床(壁)がなくなると、荷物は落下します(Fig.4)。落ちた荷物は、下に床や他の荷物がない限り、落下を続けます(Fig.5)。下に床や他の荷物があると、荷物は着地します(Fig.6)。

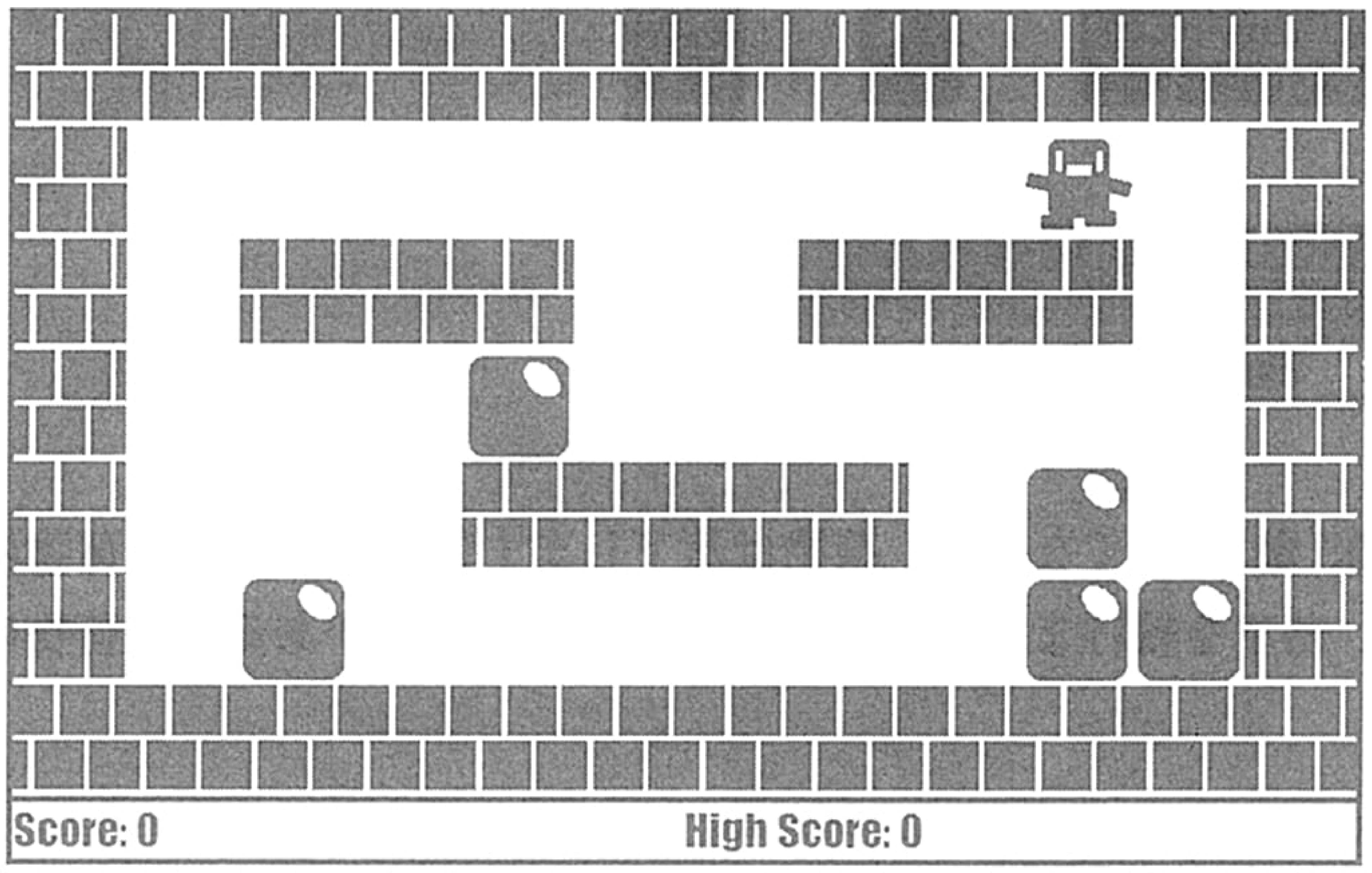
▼ **Fig.4**: 荷物が落下する



▼ **Fig.5**: 荷物が落下を続ける

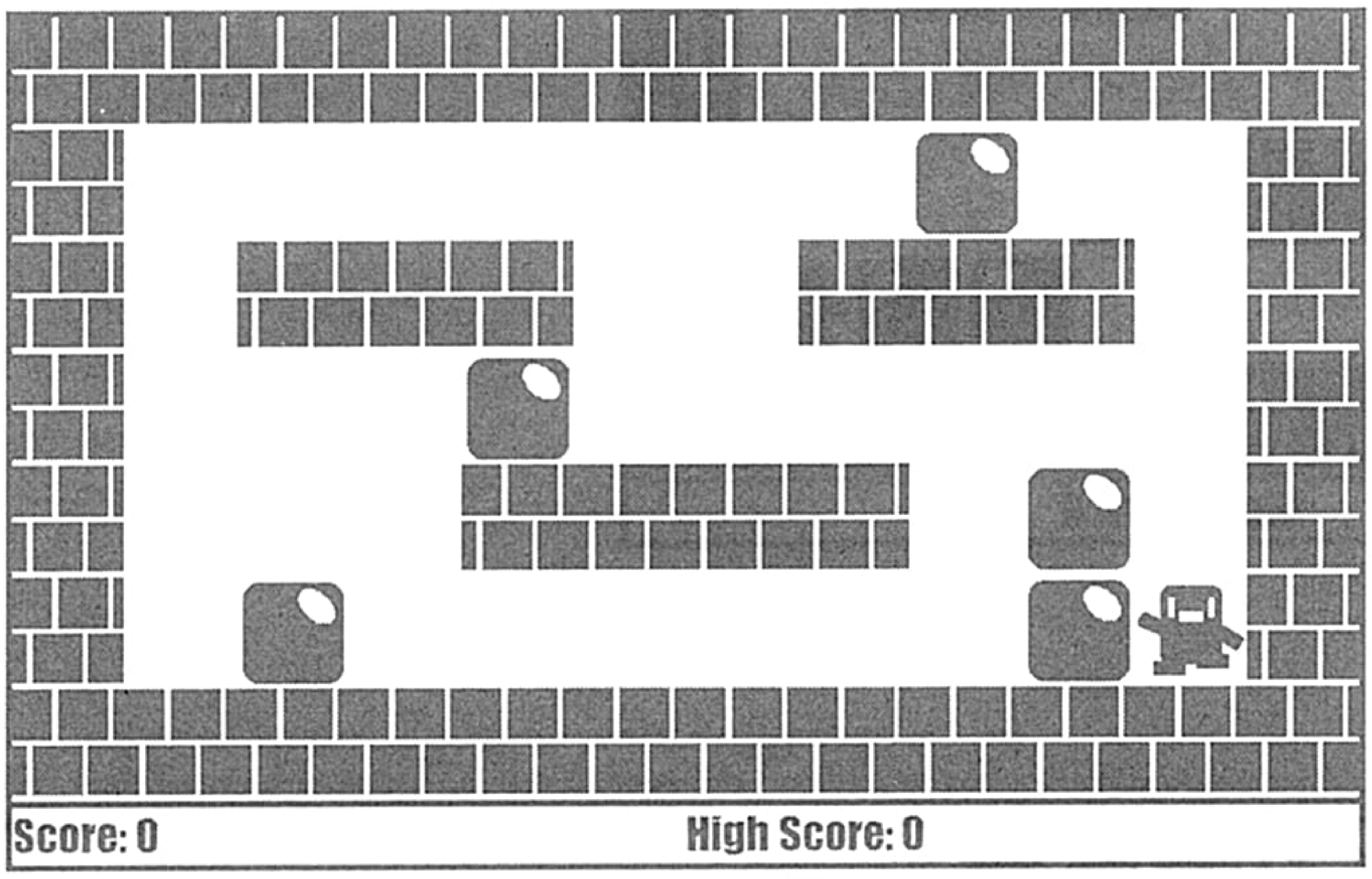


▼ **Fig.6**: 荷物が着地する



今度は荷物が2個重なっているときに、下の荷物を押してみます(Fig.7)。下の荷物が横にずれて、自機が上の荷物の下に入ります(Fig.8)。下に自機がいるので、上の荷物は落ちてきません。自機が横にずれると、支えがなくなった上の荷物は落下します(Fig.9)。

▼ Fig.7: 2個重なっている荷物を押す



▼ Fig.8: 自機で荷物を支える

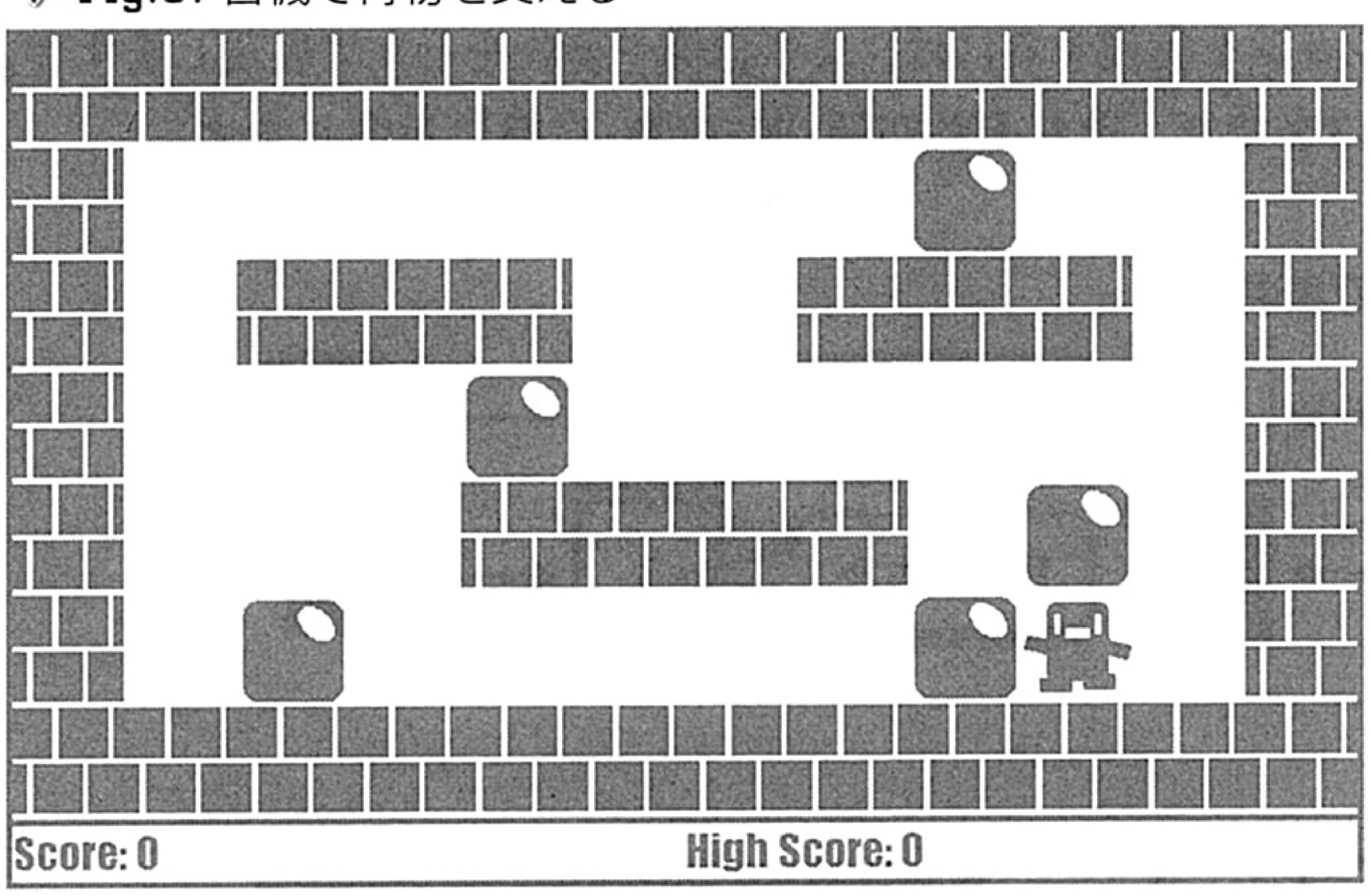
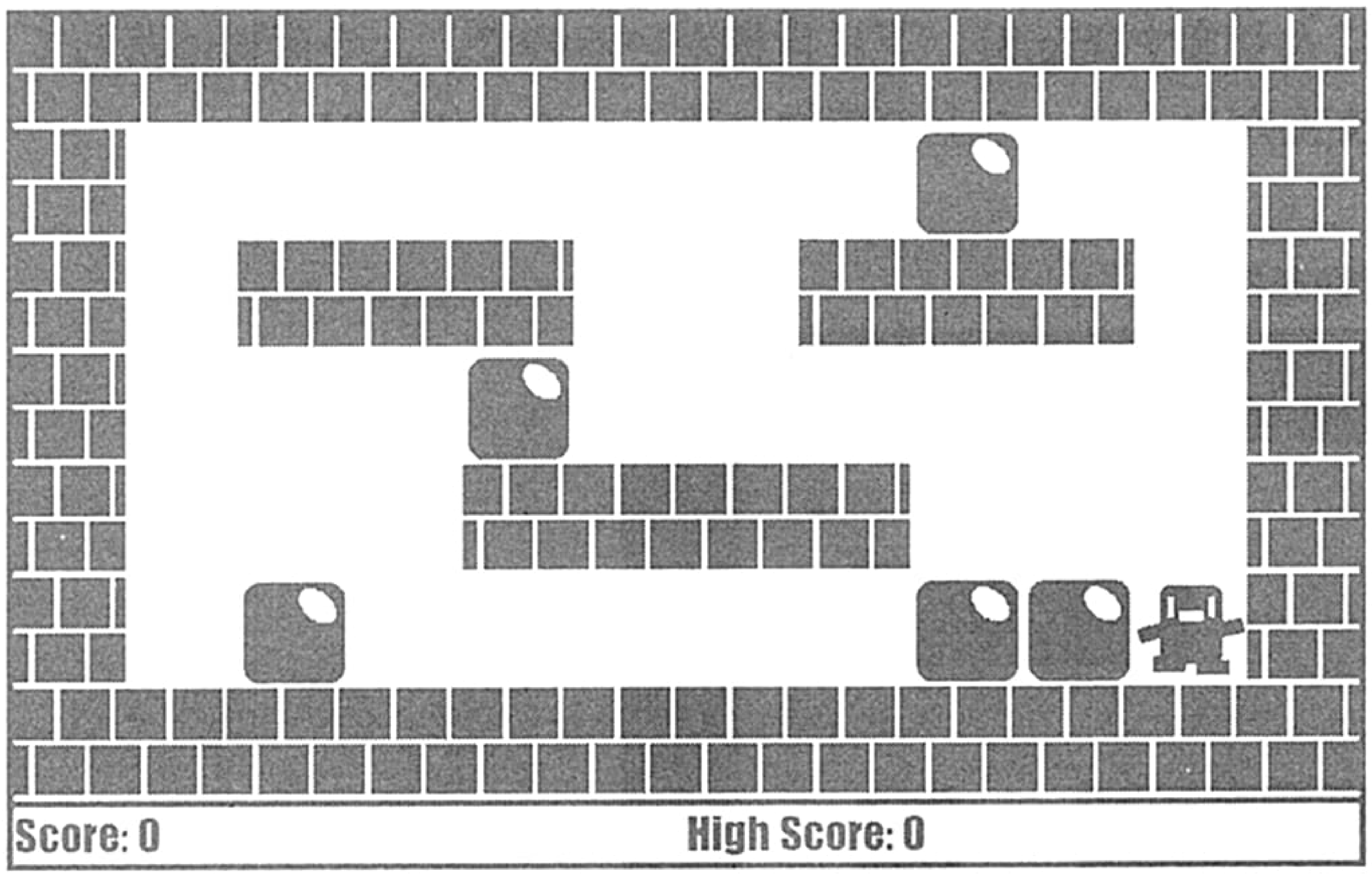


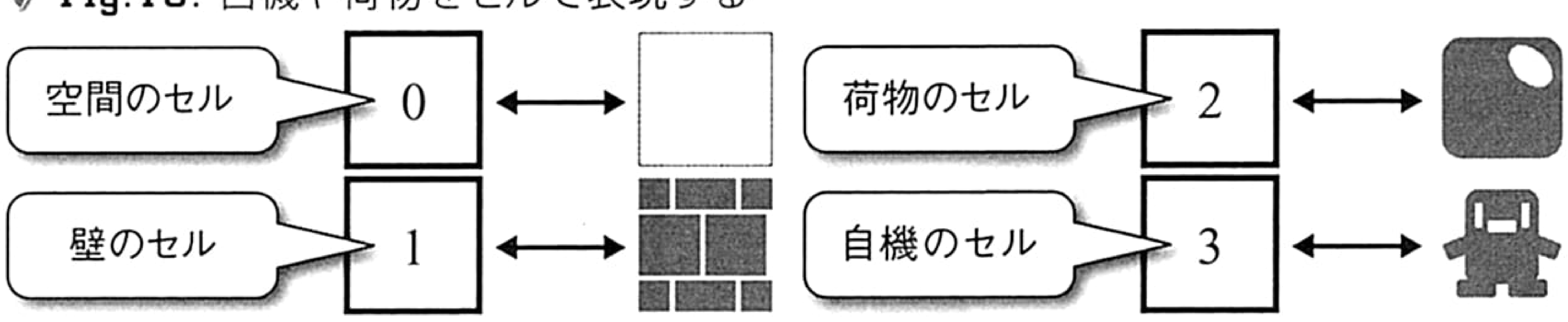
Fig.9: 自機が動くと荷物が落ちてくる



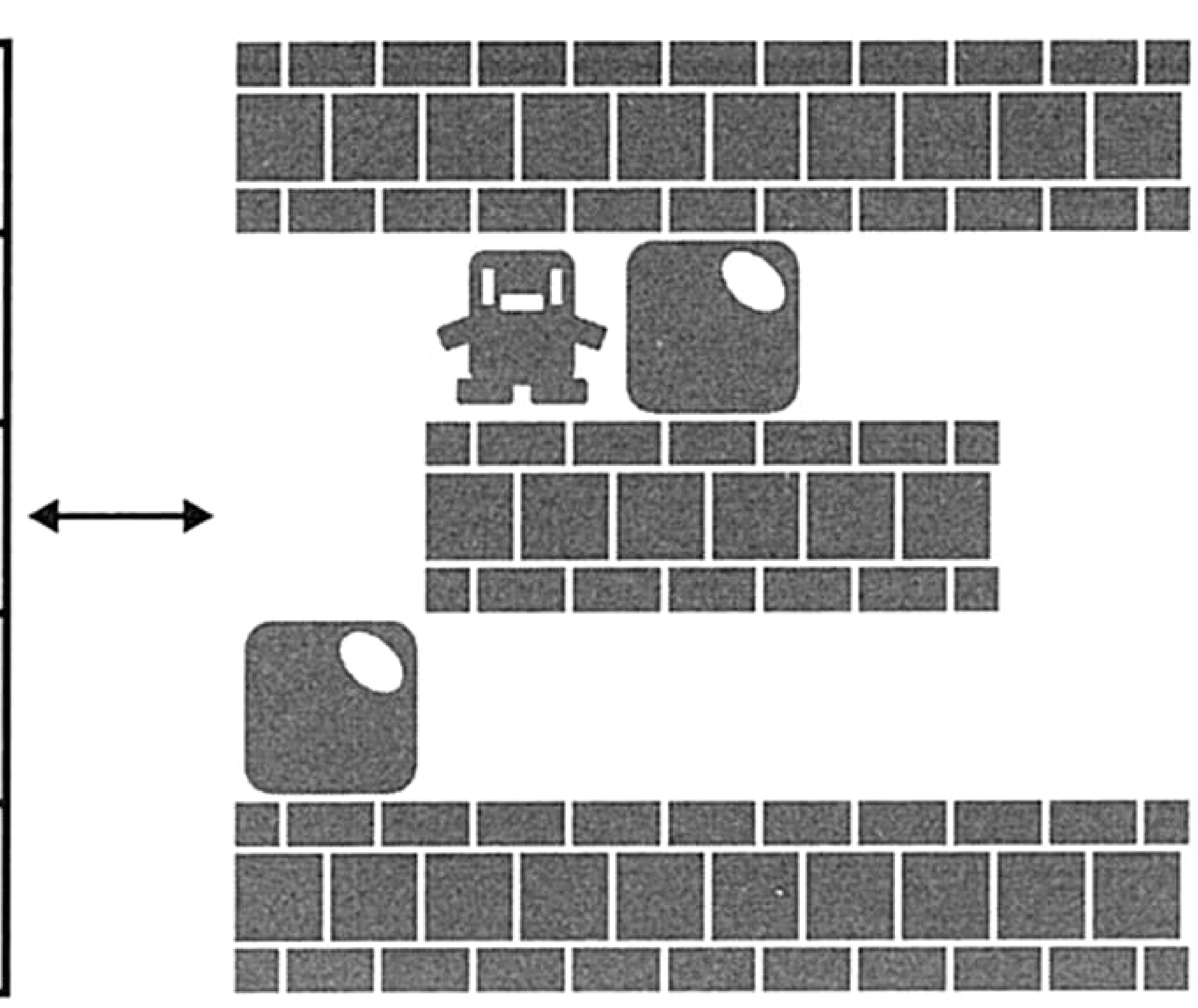
荷物を押して動かす自機

荷物を押して動かすアクションを実現するには、まず自機や荷物をセルで表現します(Fig.10)。ここではセルの種類は0から3までで、それぞれ空間・壁・荷物・自機に対応します。

Fig.10: 自機や荷物をセルで表現する

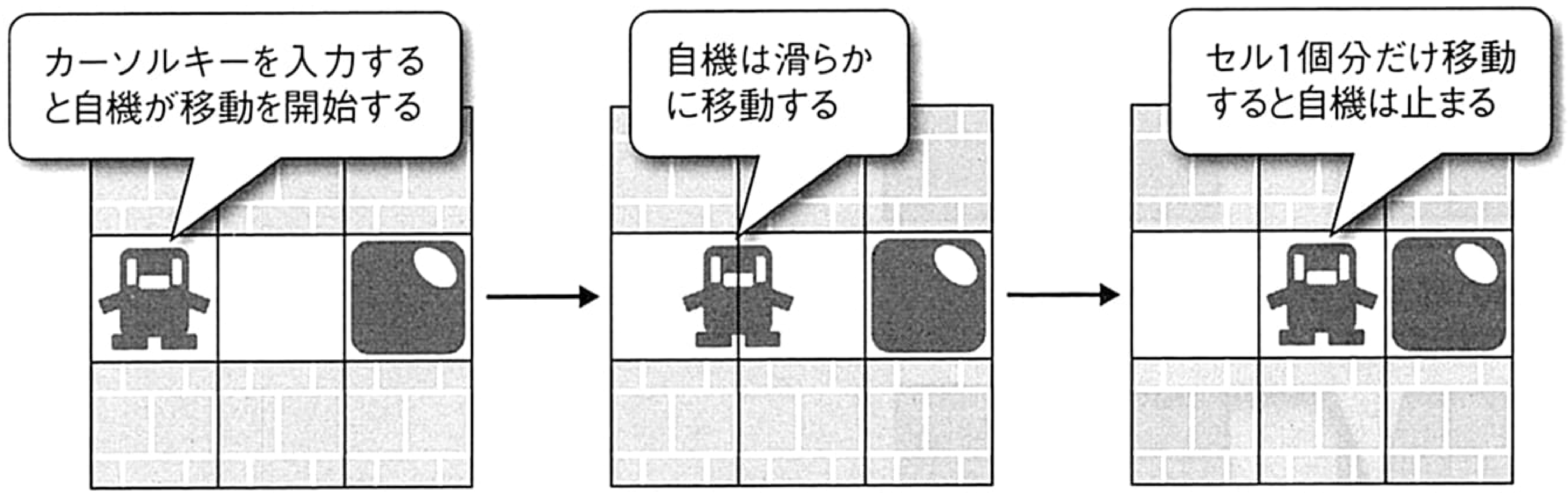


1	1	1	1	1
0	3	2	0	0
0	1	1	1	0
2	0	0	0	0
1	1	1	1	1



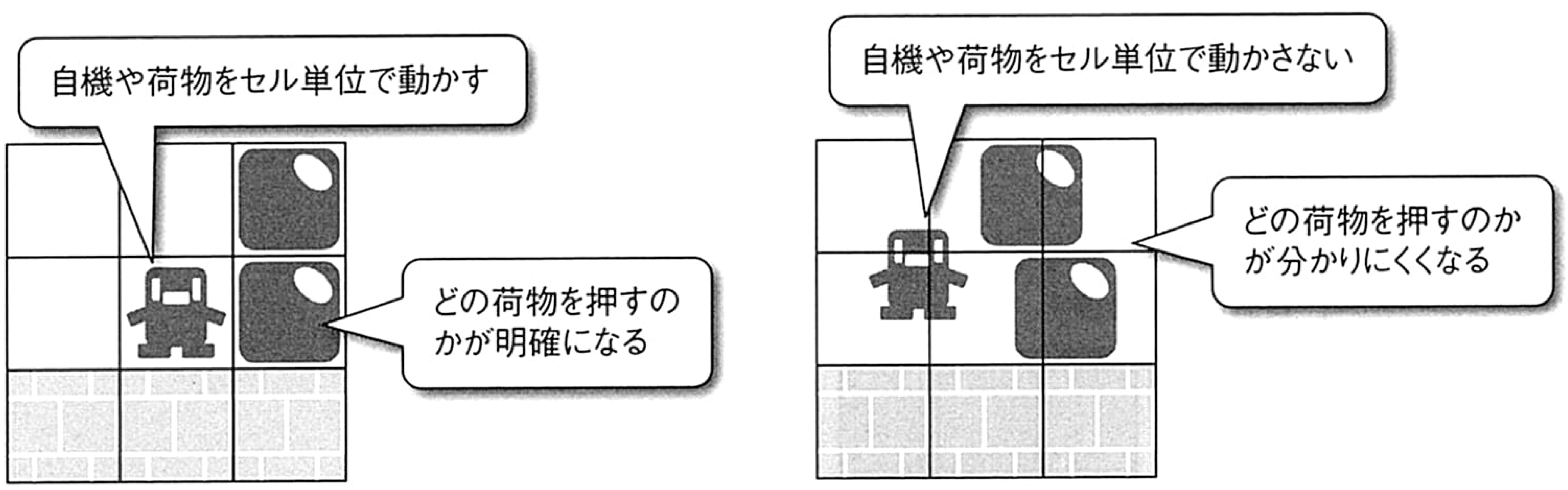
第7章・第8章のサンプルと同様に、自機はセル単位で移動します(Fig.11)。カーソルキーを入力すると、自機は滑らかに動き、セル1個分だけ移動したところで止まります。キーを押しっぱなしにすると、その方向に連続して進みます。

▼ **Fig.11**: 自機はセル単位で動く



自機はセルよりも細かい単位で動かすこともできるのですが、荷物を押して動かすゲームでは、セル単位で動かした方が操作しやすくなります(Fig.12)。あまり自機が細かく動くと、自機と荷物の位置関係が微妙になり、荷物を押しにくくなることがあるからです。「倉庫番」のようにセル1個単位で動かすか、「フラッピー」のようにセル半個単位で動かす方が、操作性がよくなります。

▼ **Fig.12**: セル単位で動かした方が操作しやすい場合

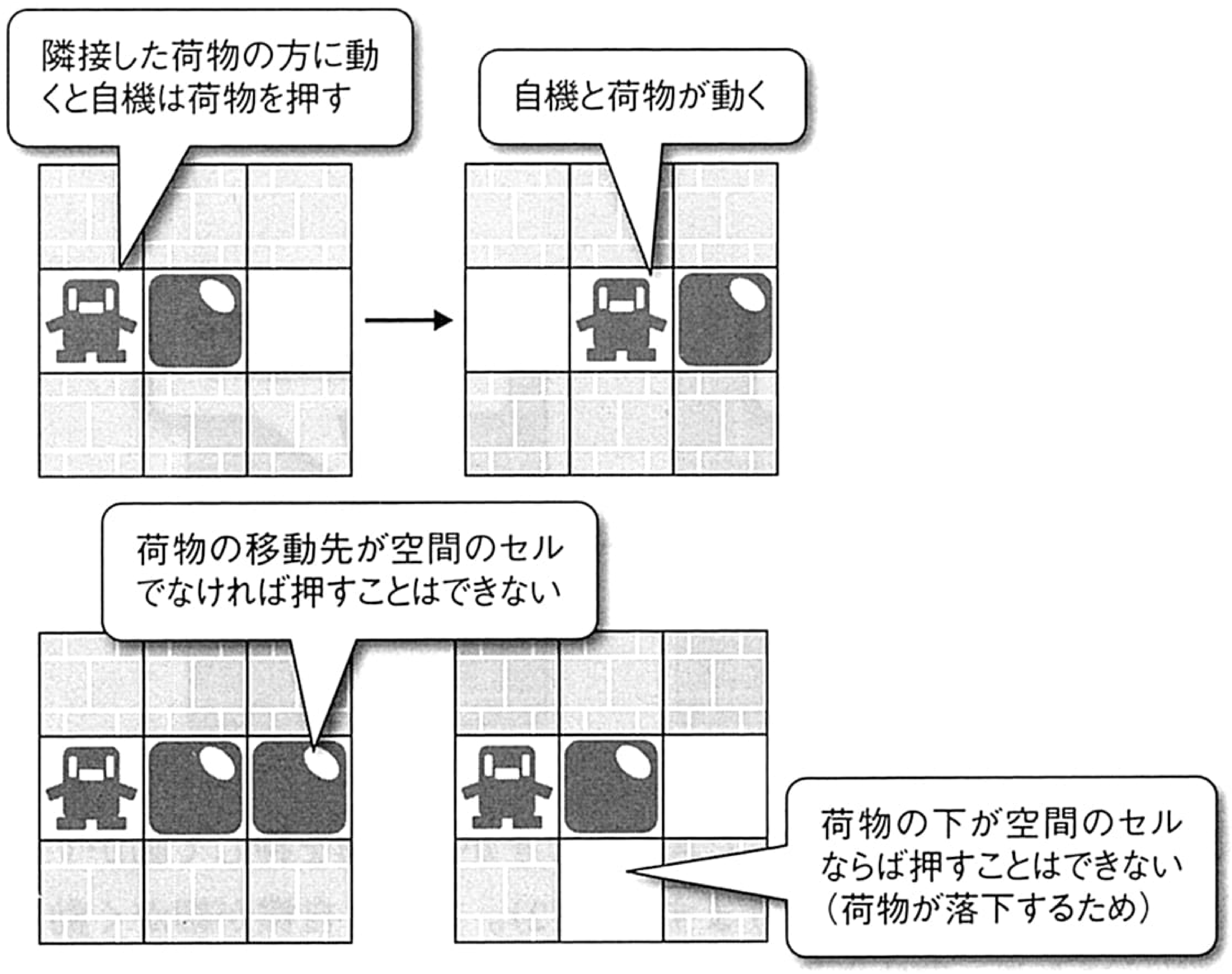


さて、自機が隣接した荷物の方向に動くと、荷物を押したことになります(Fig.13)。荷物を押すためには、

(1) 荷物の移動先が空間のセルである
(2) 荷物の下が空間のセルではない(壁・荷物・自機である)

という条件の両方が必要です。

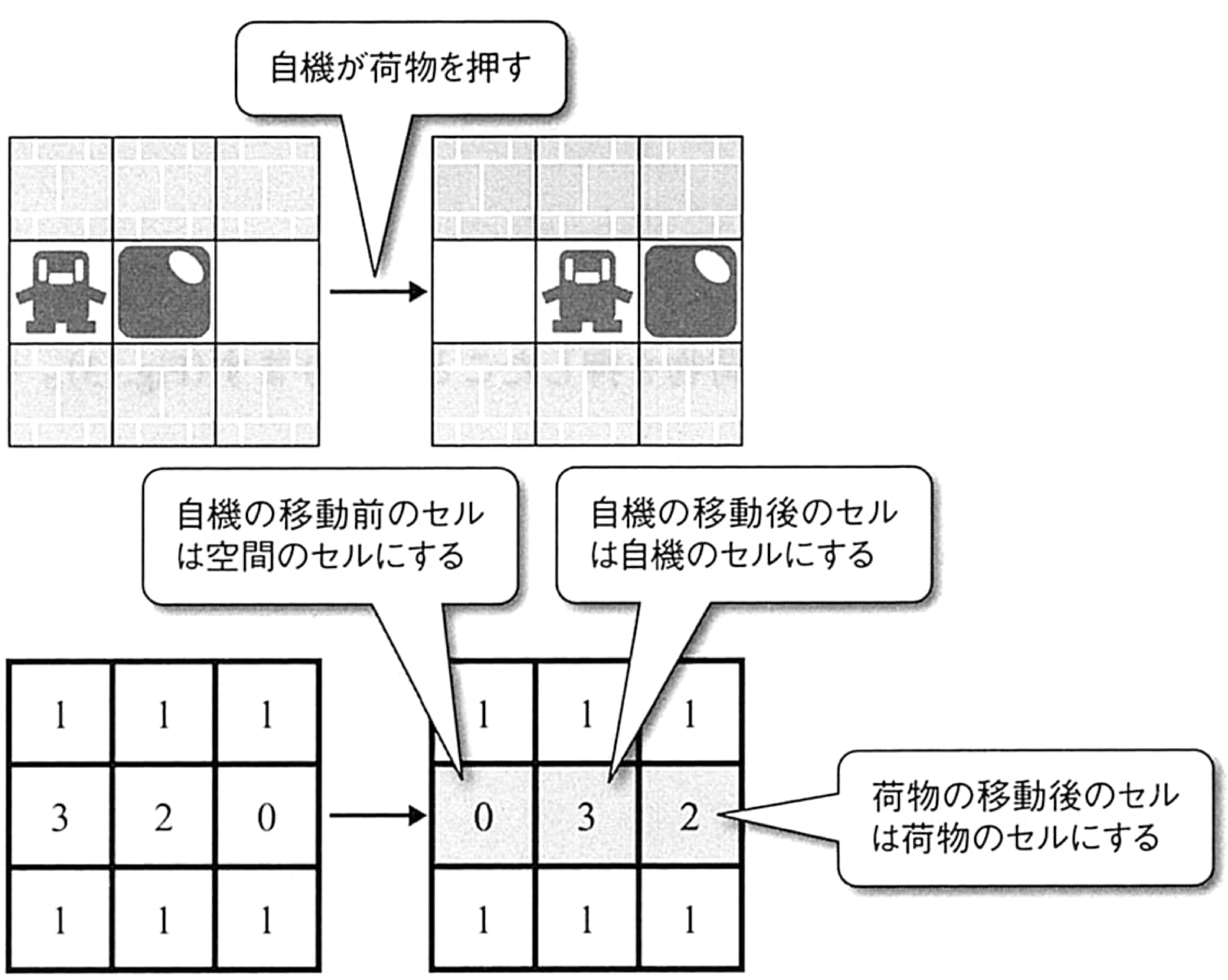
▼ **Fig.13**: 荷物を押す



(1)が満たされないと、荷物の移動先に障害物（壁・荷物）があるため、荷物は動かせません。(2)が満たされないと、荷物は支えがなく落下するので、やはり荷物は動かせません（このサンプルでは落下する荷物は押せないことにしています）。

そこで、荷物の移動先と荷物の下のセルを調べます。(1)と(2)の条件を満たしていたら、荷物を動かします（Fig.14）。

▼ **Fig.14**: 荷物を動かす



荷物を動かすときには、自機や荷物のスプライトを移動させることに加えて、セルも次のように変更します。

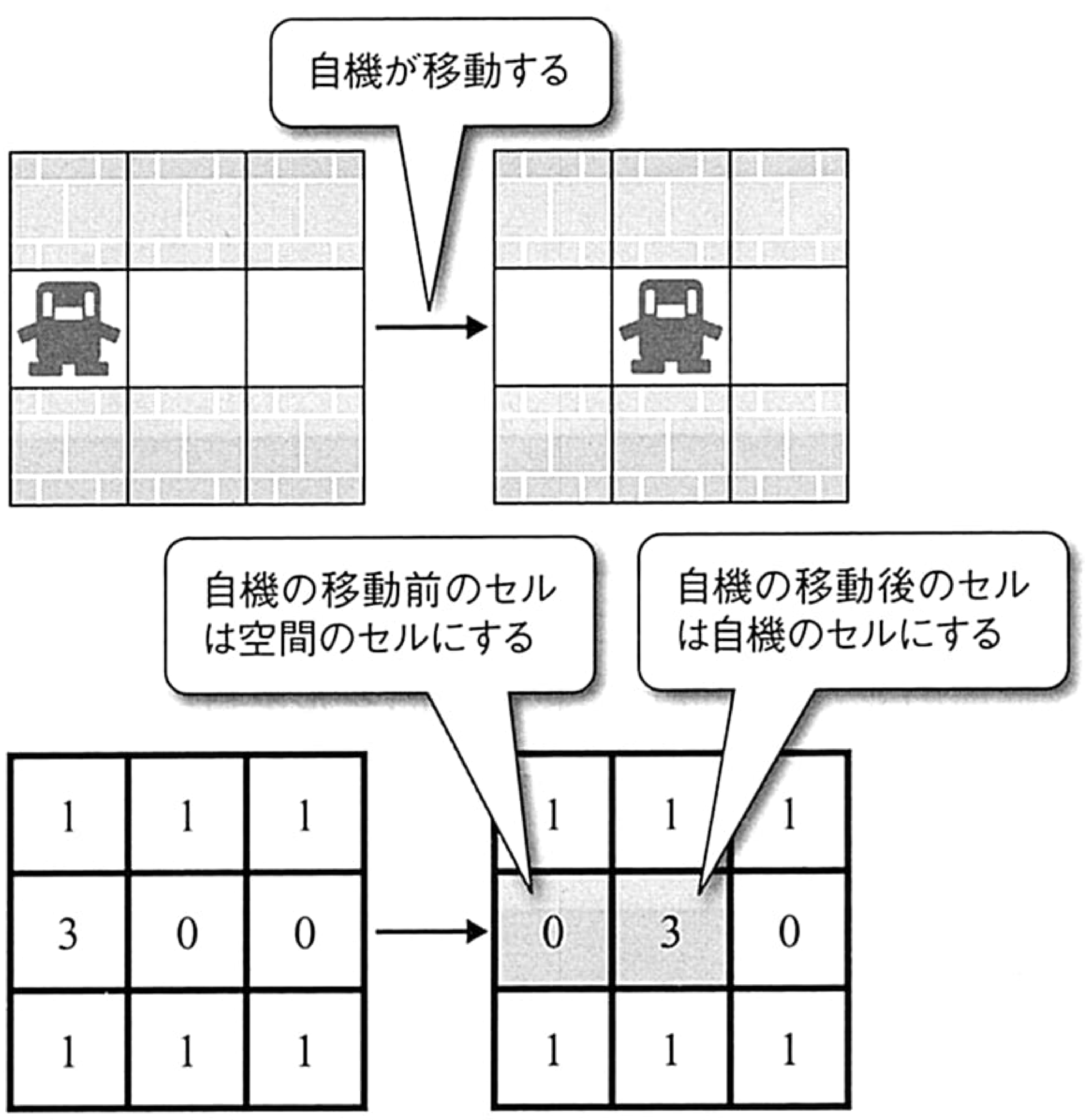
- 自機の移動前のセルを空間のセルに変更する
- 自機の移動後のセル（荷物の移動前のセル）を自機のセルに変更する
- 荷物の移動後のセルを荷物のセルに変更する

このようにセルを変更することによって、セルが画面の最新の状態を表すようにします。なお、荷物を押さずに自機だけが動くときにも、

- 自機の移動前のセルを空間のセルに変更する
- 自機の移動後のセルを自機のセルに変更する

のようにセルを変更します（Fig.15）。荷物が落下するかどうかを判定するときには、荷物の下に自機がいるかどうかも調べます。そのため、自機の最新の位置もセルに反映させておく必要があるのです。

▼ **Fig.15**: 荷物を押さずに自機が動く



荷物を押して動かす自機のプログラム

List1は自機のプログラムです。他の章と同様に、自機の処理はPlayerオブジェクトにまとめました。

まず、自機をセル単位で動かすために、

```
if (x==Math.floor(x) && y==Math.floor(y)) {
```

のように、自機の座標が整数かどうかを調べます。整数のとき、つまり自機の位置がセルにぴったり合っているときには、

```
vx=vy=0;
if (game.isKey(KEY_LEFT)) vx=-1; else
if (game.isKey(KEY_RIGHT)) vx=1; else
if (game.isKey(KEY_UP)) vy=-1; else
if (game.isKey(KEY_DOWN)) vy=1;
```

のように、カーソルキーの入力に応じて、自機の移動方向(vx, vy)を決めます。

自機を動かすには、自機の移動先のセルを調べます。自機の移動先が、

```
if (field.getCell(x+vx, y+vy)==CELL_VOID) {
```

のように空間のセルのときには、

```
field.setCell(x, y, CELL_VOID);
field.setCell(x+vx, y+vy, CELL_PLAYER);
```

として自機を移動させます。自機の移動前のセルを空間に、移動後のセルを自機にします。

一方、自機の移動先が荷物のときには、

```
if (
    vy==0 &&
    field.getCell(x+vx, y)==CELL_LOAD &&
    field.getCell(x+vx+vx, y)==CELL_VOID &&
    field.getCell(x+vx, y+1)!=CELL_VOID
) {
```

のように、荷物の移動先が空間かどうか、荷物の下が空間以外かどうかも調べます。また、このサンプルでは荷物を上下には押せないことにしたので、自機の移動が横方向かどうかも調べています(vy==0)。

荷物が動かせそうなときには、押そうとしている荷物のオブジェクトを探します。これは、

```
for (i=0; i<load.length; i++) {
    if (
        load[i].sprite.getState()==STATE_LOAD_STAY &&
        load[i].sprite.getX()==x+vx &&
        load[i].sprite.getY()==y+vy
    ) {
```

のようなfor文とif文を使います。自機の移動先の座標と荷物の座標を比べて、荷物が静止しており（STATE_LOAD_STAY）、かつ座標が一致したら、その荷物を動かします。荷物を動かすには、

```
field.setCell(x, y, CELL_VOID);
field.setCell(x+vx, y, CELL_PLAYER);
field.setCell(x+vx+vx, y, CELL_LOAD);
```

のようにセルを書き換えます。自機の移動前のセルは空間に、自機の移動後のセルは自機に、荷物の移動後のセルは荷物にします。また、

```
load[i].vx=vx;
load[i].vy=vy;
load[i].sprite.setState(STATE_LOAD_MOVE);
```

のように、荷物の速度と状態を設定します。これは荷物のスプライトを移動させるためです。

もしも条件に該当する荷物が見つからなかったら、

```
if (i==load.length) {
    vx=vy=0;
}
```

のように自機の移動を中止します。これは押そうとした荷物が移動中か落下中のときに起こります。

List 1: 荷物を押して動かす自機（pushLoad.js）

```
// グリッド(座標を補正するための定数)
var GRID=5;

// 自機のスピード
var SPEED=1.0/GRID;

// 自機
```

```
function Player(x, y) {
  var obj=this;

  // スプライトの作成
  obj.sprite=game.createSprite();

  // アニメーションとグリッドの設定
  obj.sprite.initAnim(playerWalkAnim);
  obj.sprite.setGrid(GRID);

  // 座標と速度の設定
  obj.sprite.setXY(x, y);
  obj.vx=0;
  obj.vy=0;

  // 更新処理
  obj.sprite.onUpdate=function() {

    // 座標と速度の取得
    var x=obj.sprite.getX(), y=obj.sprite.getY();
    var vx=obj.vx, vy=obj.vy;
    var i;

    // 自機の座標が整数のとき
    if (x==Math.floor(x) && y==Math.floor(y)) {

      // カーソルキーの入力に応じて上下左右に移動する
      vx=vy=0;
      if (game.isKey(KEY_LEFT)) vx=-1; else
      if (game.isKey(KEY_RIGHT)) vx=1; else
      if (game.isKey(KEY_UP)) vy=-1; else
      if (game.isKey(KEY_DOWN)) vy=1;

      // 移動先が空間のときには移動する
      if (field.getCell(x+vx, y+vy)==CELL_VOID) {

        // 移動前のセルを空間にする
        field.setCell(x, y, CELL_VOID);

        // 移動後のセルを自機にする
        field.setCell(x+vx, y+vy, CELL_PLAYER);
      } else

      // 移動先が荷物で、荷物の移動先が空間で、
      // 荷物の下が空間ではないときには荷物を押す
      if (
        vy==0 &&
        field.getCell(x+vx, y)==CELL_LOAD &&
```

```
      field.getCell(x+vx+vx, y)==CELL_VOID &&
      field.getCell(x+vx, y+1)!=CELL_VOID
    ) {
      // 押そうとしている荷物を探す
      for (i=0; i<load.length; i++) {

        // 静止していて座標が一致する荷物を探す
        if (
          load[i].sprite.getState()==STATE_LOAD_STAY &&
          load[i].sprite.getX()==x+vx &&
          load[i].sprite.getY()==y+vy
        ) {
          // 自機の移動前のセルを空間にする
          field.setCell(x, y, CELL_VOID);

          // 自機の移動後のセルを自機にする
          field.setCell(x+vx, y, CELL_PLAYER);

          // 荷物の移動後のセルを荷物にする
          field.setCell(x+vx+vx, y, CELL_LOAD);

          // 荷物の速度と状態を設定する
          load[i].vx=vx;
          load[i].vy=vy;
          load[i].sprite.setState(STATE_LOAD_MOVE);
          break;
        }
      }

      // 該当する荷物が見つからなかったら移動をやめる
      if (i==load.length) {
        vx=vy=0;
      }
    } else

    // 移動先が空間ではなく、押せる荷物でもないときには、
    // 移動をやめる
    {
      vx=vy=0;
    }
  }

  // 座標と速度の更新
  obj.sprite.setXY(x+vx*SPEED, y+vy*SPEED);
  obj.vx=vx;
  obj.vy=vy;

  // 移動中はアニメーションを再生する
```

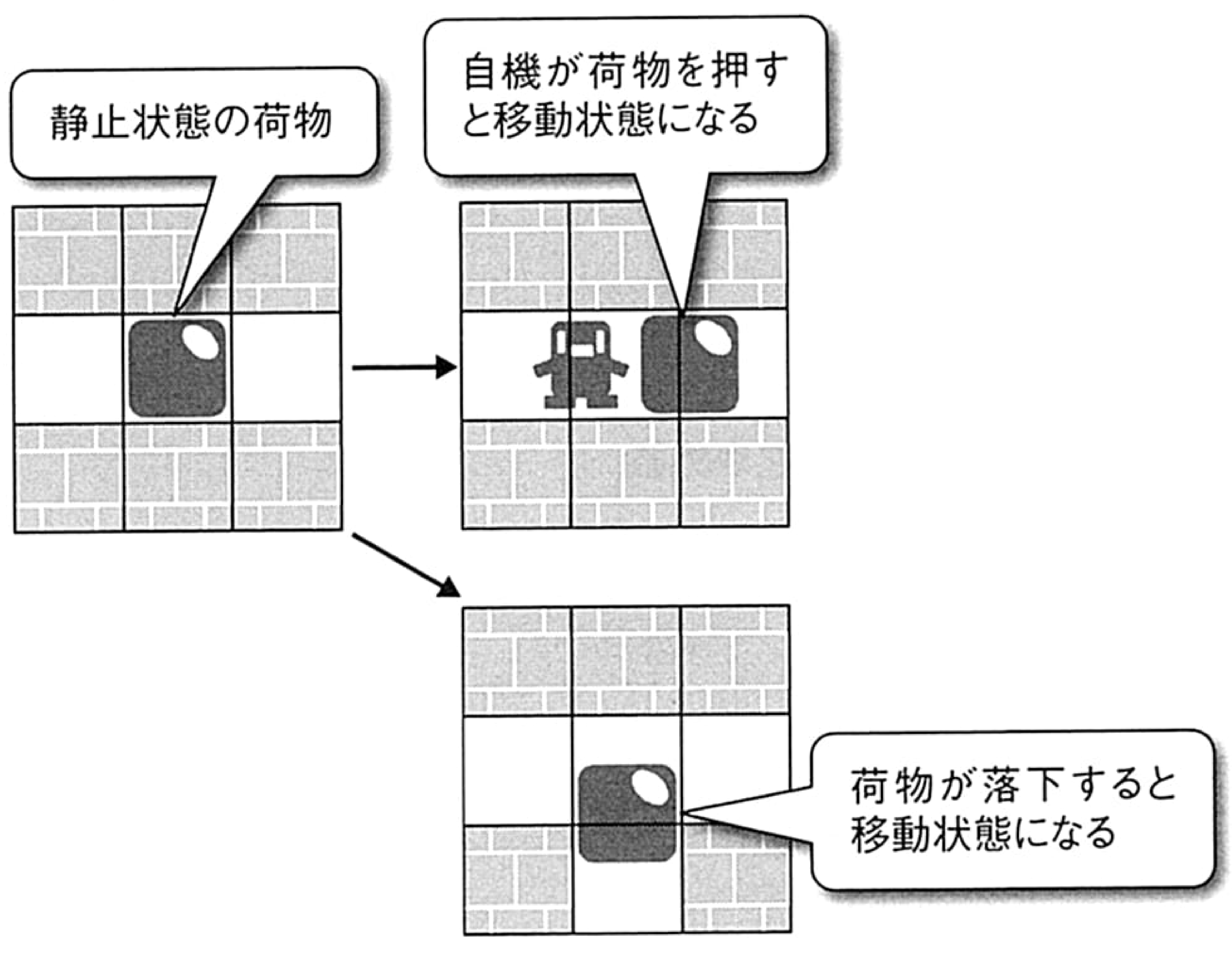
```
    if (vx!=0 || vy!=0) {
      obj.sprite.stepAnim();
    }
  }
}
```

押すと動く荷物

荷物を押して動かすための処理は、ほとんど自機側で行います(p.259)。荷物側で行うのは、荷物の下に何もないときに落下する処理と、時間とともに荷物を滑らかに移動させる処理です。

荷物には静止状態と移動状態を設けます(Fig.16)。静止状態のときに自機が荷物を押したら、移動状態に移行します。また、荷物の下のセルが空間のときには、荷物は支えがなく落下するため、やはり移動状態に移行します。

▼ **Fig.16**: 静止状態の荷物

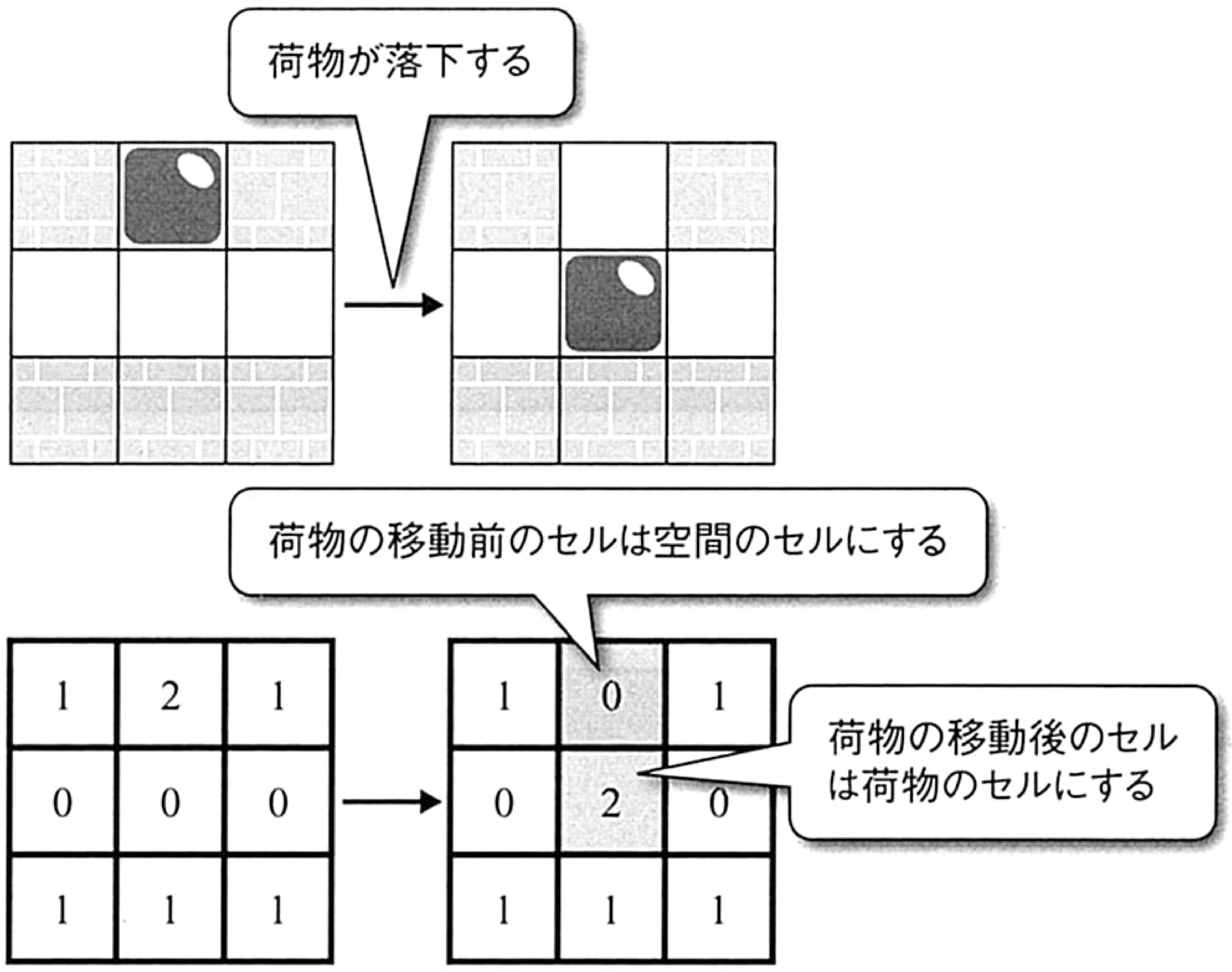


荷物が落下するときには、

- 荷物の移動前のセルを空間のセルに変更する
- 荷物の移動後のセルを荷物のセルに変更する

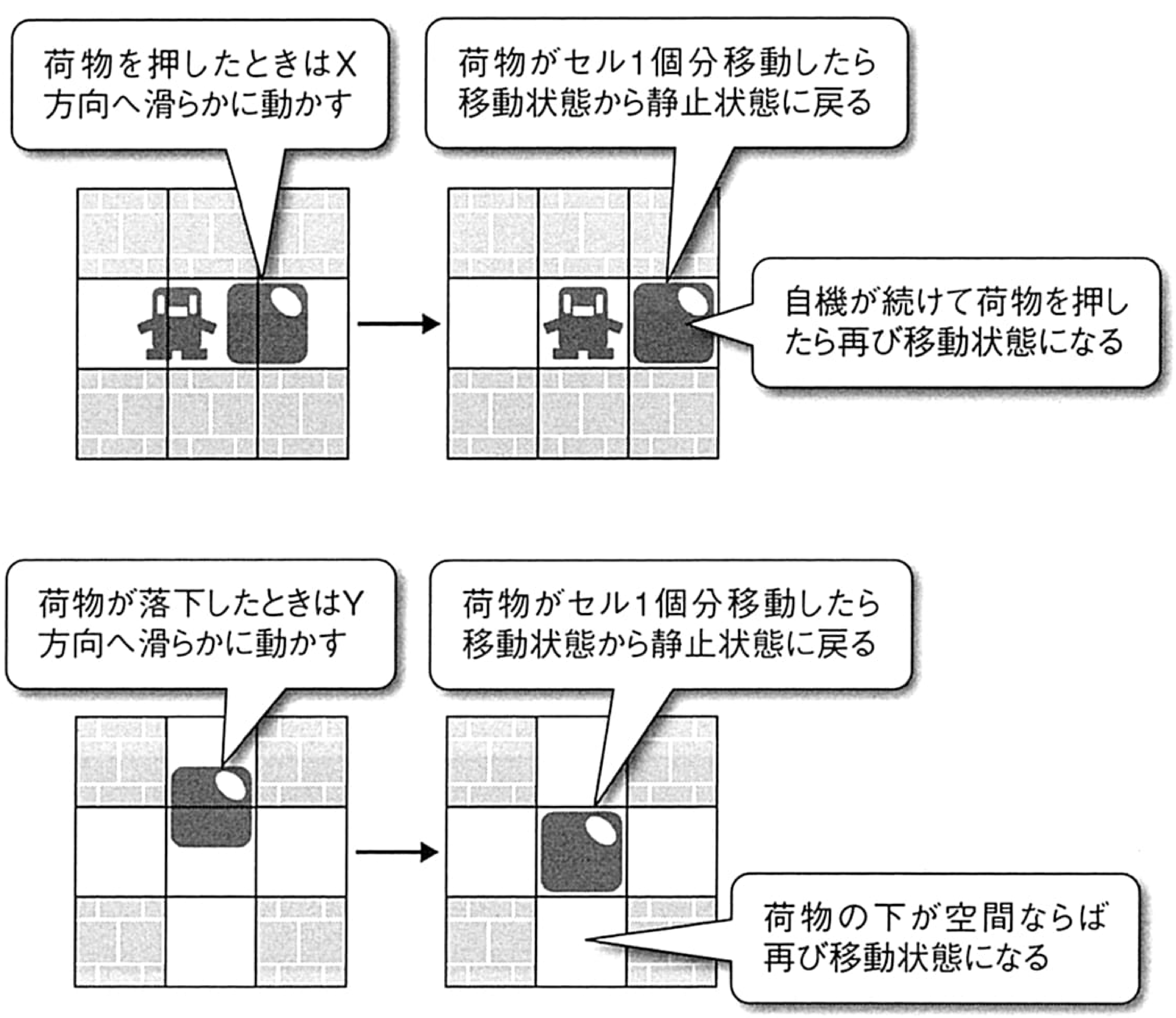
のように、セルを変更します(Fig.17)。これは自機が移動するときと同じ要領です(p.261)。

▼ Fig.17: 荷物の落下



移動状態では、荷物を押したときにはX方向に、落下したときにはY方向に、それぞれ荷物を滑らかに動かします(Fig.18)。荷物の座標が整数のとき、つまり荷物がセル1個分移動したときには、静止状態に戻ります。

▼ Fig.18: 移動状態の荷物



押すと動く荷物のプログラム

List2は荷物のプログラムです。荷物の処理はLoadオブジェクトにまとめました。
荷物が静止状態のときには、

```
if (field.getCell(x, y+1)==CELL_VOID) {
```

のように、荷物の下のセルが空間かどうかを調べます。空間のときには、荷物を落下させるために、

```
vy=1;
```

のように下方向の速度を設定し、

```
field.setCell(x, y, CELL_VOID);
field.setCell(x, y+vy, CELL_LOAD);
```

のようにセルを変更します。移動前のセルは空間に、移動後のセルは荷物にします。そして、

```
obj.sprite.setState(STATE_LOAD_MOVE);
```

のように、移動状態に移行します。
荷物が移動状態のときには、

```
obj.sprite.setXY(x+vx*SPEED, y+vy*SPEED);
x=obj.sprite.getX();
y=obj.sprite.getY();
```

のように座標を更新して、荷物を滑らかに動かします。荷物の座標が整数のとき、つまり荷物がセル1個分移動したときには、

```
if (x==Math.floor(x) && y==Math.floor(y)) {
    vx=vy=0;
    obj.sprite.setState(STATE_LOAD_STAY);
}
```

のように、速度を0にして、静止状態に戻ります。

List2: 押すと動く荷物(pushLoad.js)

```
// 荷物の状態(静止・移動)
var STATE_LOAD_STAY=1;
var STATE_LOAD_MOVE=2;

// 荷物
function Load(x, y) {
  var obj=this;

  // スプライトの作成
  obj.sprite=game.createSprite();

  // アニメーション・グリッド・状態の設定
  obj.sprite.initAnim(loadAnim);
  obj.sprite.setGrid(GRID);
  obj.sprite.setState(STATE_LOAD_STAY);

  // 座標と速度の設定
  obj.sprite.setXY(x, y);
  obj.vx=0;
  obj.vy=0;

  // 更新処理
  obj.sprite.onUpdate=function() {

    // 座標と速度の取得
    var x=obj.sprite.getX(), y=obj.sprite.getY();
    var vx=obj.vx, vy=obj.vy;

    // 静止状態
    if (obj.sprite.getState()==STATE_LOAD_STAY) {

      // 荷物の下に何もないとき
      if (field.getCell(x, y+1)==CELL_VOID) {

        // 下方向の速度を設定する
        vy=1;

        // 移動前のセルを空間にする
        field.setCell(x, y, CELL_VOID);

        // 移動後のセルを荷物にする
        field.setCell(x, y+vy, CELL_LOAD);

        // 移動状態に移行する
        obj.sprite.setState(STATE_LOAD_MOVE);
      }
```

```
    }

    // 移動状態
    if (obj.sprite.getState()==STATE_LOAD_MOVE) {

      // 座標の更新
      obj.sprite.setXY(x+vx*SPEED, y+vy*SPEED);
      x=obj.sprite.getX();
      y=obj.sprite.getY();

      // 荷物の座標が整数のとき
      if (x==Math.floor(x) && y==Math.floor(y)) {

        // 速度の初期化
        vx=vy=0;

        // 静止状態に移行する
        obj.sprite.setState(STATE_LOAD_STAY);
      }
    }

    // 速度の更新
    obj.vx=vx;
    obj.vy=vy;
  }
}
```

荷物を押して動かすゲームの本体

List3は荷物を押して動かすゲームの本体です。ゲームの初期化処理(init関数)とメインルーチン(main関数)を定義します。

初期化処理では、

```
for (i=0, y=0; y<field.getYCount(); y++) {
    for (x=0; x<field.getXCount(); x++) {
```

のようなfor文を使って、全てのセルを調べます。そして、自機と荷物のセルに関しては、

```
switch (field.getCell(x, y)) {
    case CELL_PLAYER:
```

```
        player=new Player(x, y);
        break;
    case CELL_LOAD:
        load[i++]=new Load(x, y);
        break;
}
```

のように、それぞれ自機と荷物のオブジェクトを生成します。自機と荷物の初期位置は座標で指定してもよいのですが(p.219)、このようにセルを調べて位置を決める方法もあります。この方法の利点は、セルのデータに自機や荷物の位置を組み込めるので、作業が簡単で、間違いも少ないことです。

List3: 荷物を押して動かすゲームの本体(pushLoad.js)

```
// セルの種類(空間・壁・荷物・自機)
var CELL_VOID=0;
var CELL_WALL=1;
var CELL_LOAD=2;
var CELL_PLAYER=3;

// ゲーム本体・自機・荷物・フィールドのオブジェクト
var game;
var player;
var load;
var field;

// 自機と荷物のアニメーション
var playerWalkAnim;
var loadAnim;

// 初期化
function init() {
  var x, y, i;

  // 画面の消去
  game.clear();

  // フィールドの作成
  field=game.createField();

  // セルの種類に対応する画像の一覧を配列で設定する
  field.initCellImage(new Array(
    "void.png", "wall.png",
    "void.png", "void.png"
  ));

  // セルの初期化
```

```
  field.initCell(new Array(
    "111111111111",
    "100300002001",
    "101110011101",
    "100020000001",
    "100011110201",
    "102000000201",
    "111111111111"
  ));

  // 自機と荷物を生成する
  load=new Array();
  for (i=0, y=0; y<field.getYCount(); y++) {
    for (x=0; x<field.getXCount(); x++) {
      switch (field.getCell(x, y)) {

        // 自機の生成
        case CELL_PLAYER:
          player=new Player(x, y);
          break;

        // 荷物の生成
        case CELL_LOAD:
          load[i++]=new Load(x, y);
          break;
      }
    }
  }
}

// メインルーチン
function main() {

  // ゲーム本体の作成
  game=new Game();

  // アニメーションの作成
  playerWalkAnim=new Array(
    "pcWalk0.png", "pcWalk2.png", "pcWalk0.png",
    "pcWalk1.png", "pcWalk3.png", "pcWalk1.png"
  );
  loadAnim=new Array("load.png");

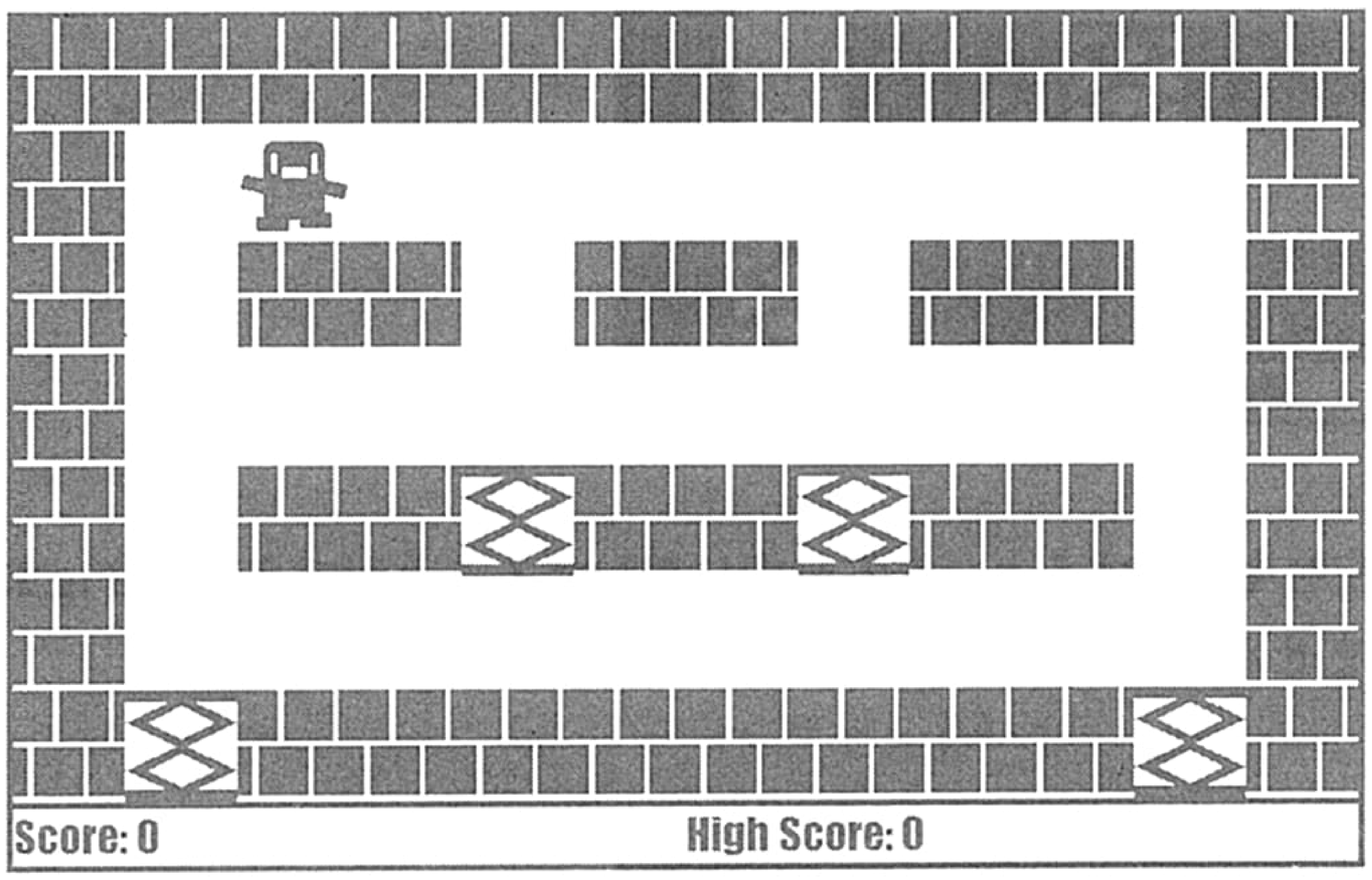
  // ゲームの初期化
  init();
}
```

トランポリン

次に紹介するのは、トランポリンを使ったアクションです。トランポリンも数多くのゲームに採用されていますが、ここでは「マッピー」(ナムコ 1983年)の動きを再現してみました。トランポリンのサンプルを実行するには、trampoline.htmlをブラウザで開きます。

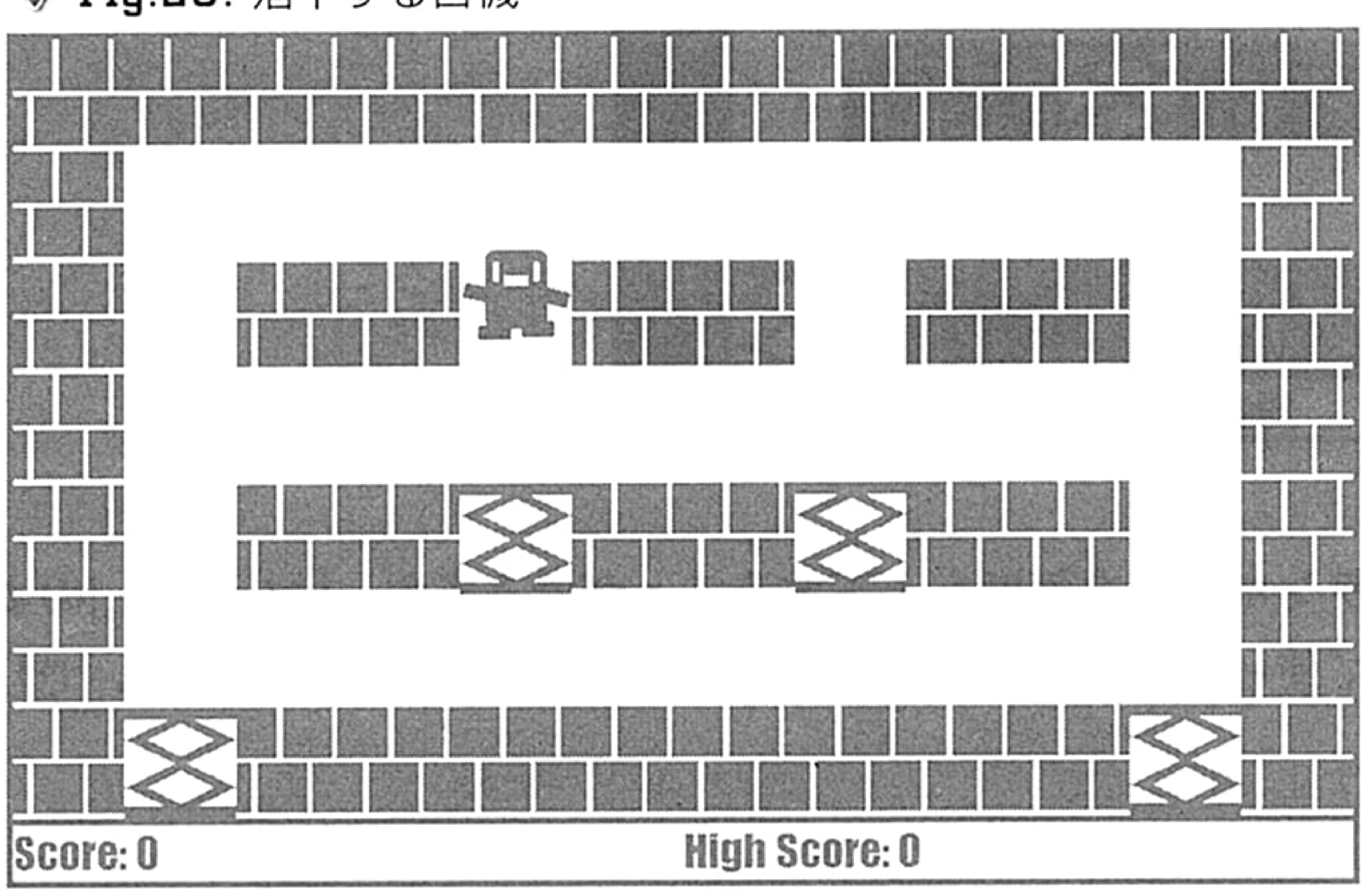
自機は画面左上に表示されます(Fig.19)。カーソルキーで自機は左右に動きます。

▼ **Fig.19**: トランポリンのサンプル



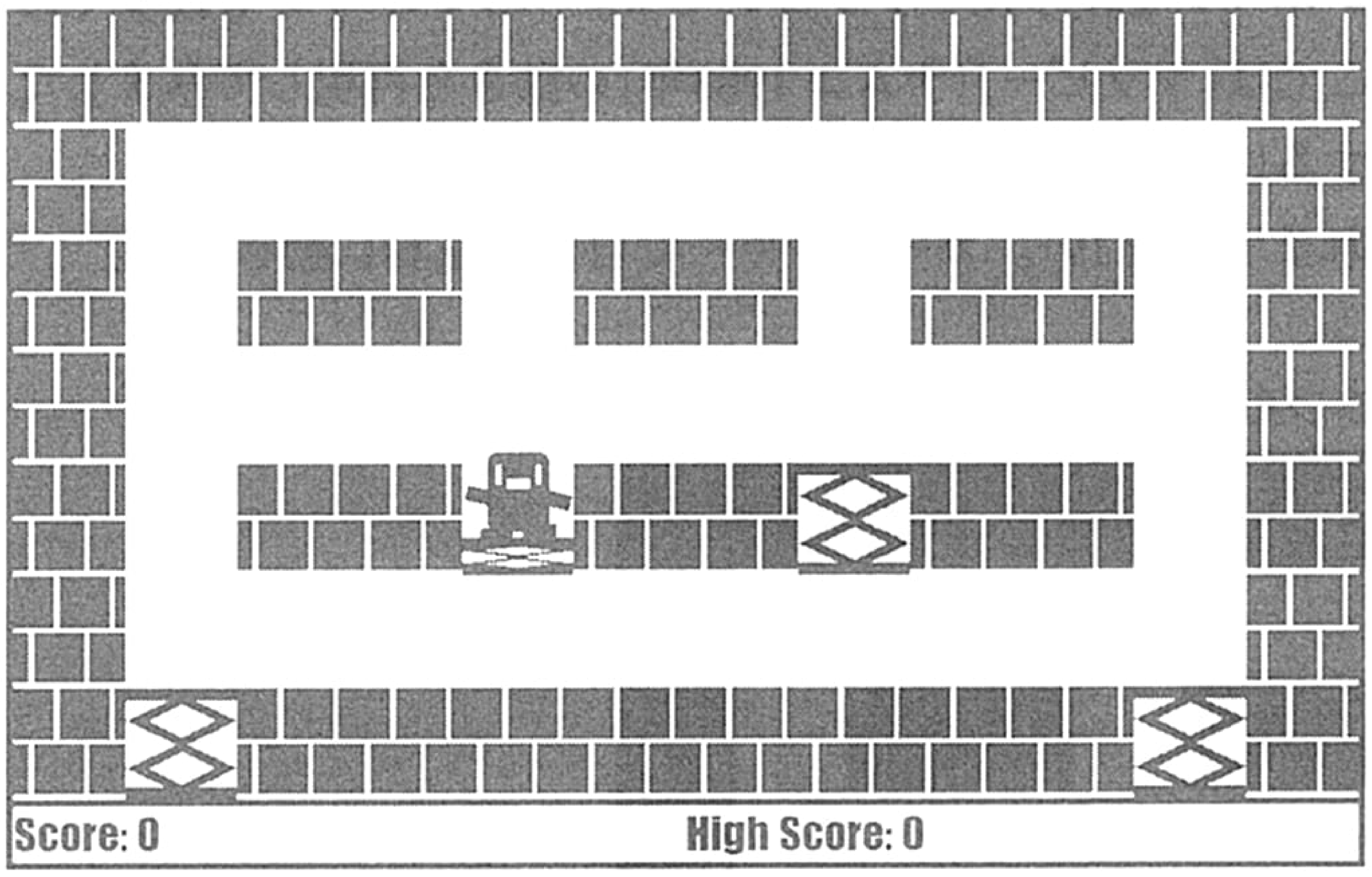
フィールドにはトランポリンが配置されています。トランポリンがある穴に入ると、自機は落下します(Fig.20)。

▼ **Fig.20**: 落下する自機



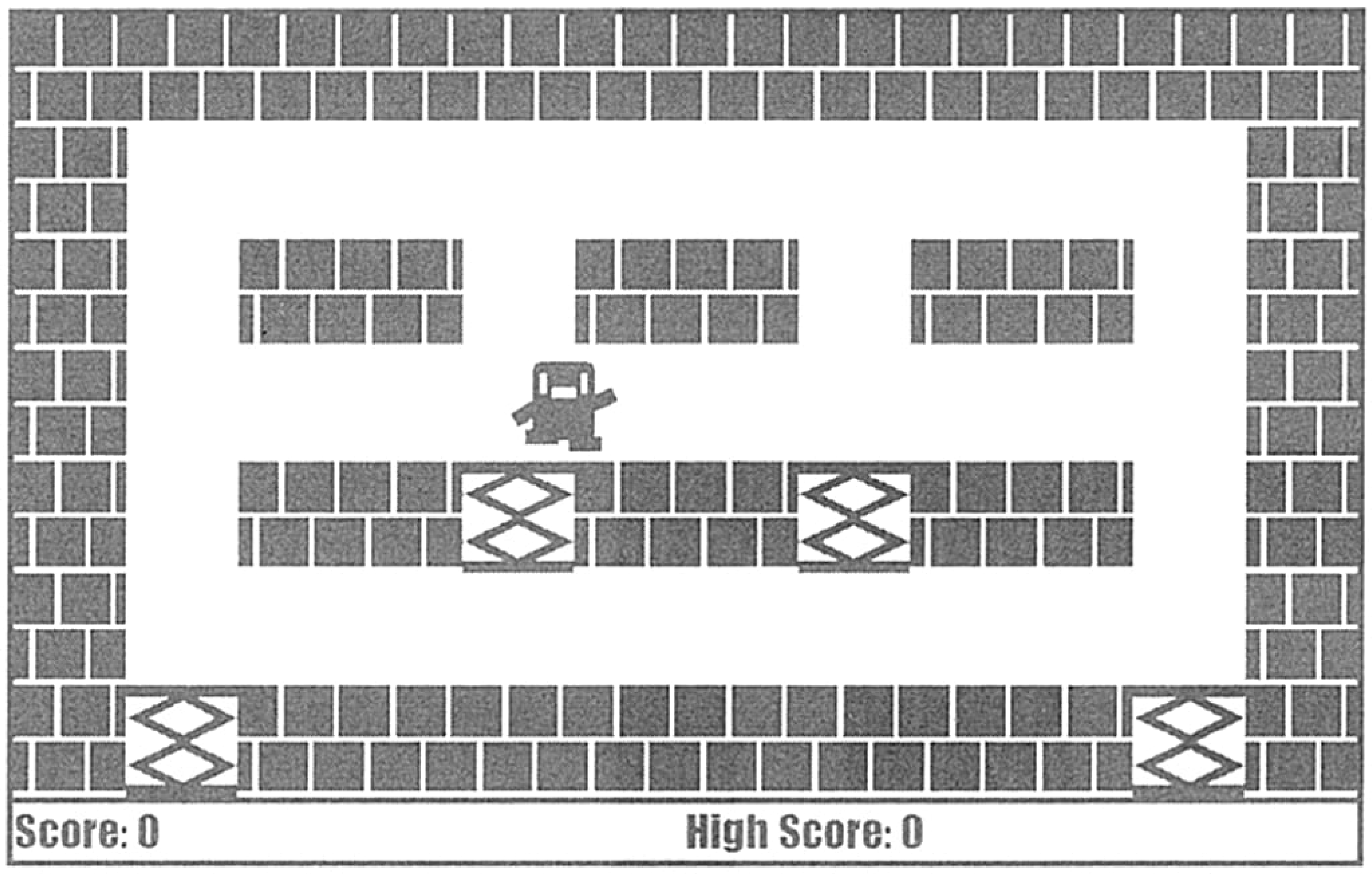
自機がトランポリンに乗ると、トランポリンは収縮します(Fig.21)。トランポリンが元の形に戻ると、その勢いで自機は上昇します。

▼ **Fig.21**: 収縮するトランポリン



上昇中にカーソルキーの左右を入力すると、左右の床に着地することができます(Fig.22)。下降中は着地することができません。

▼ **Fig.22**: 着地する自機

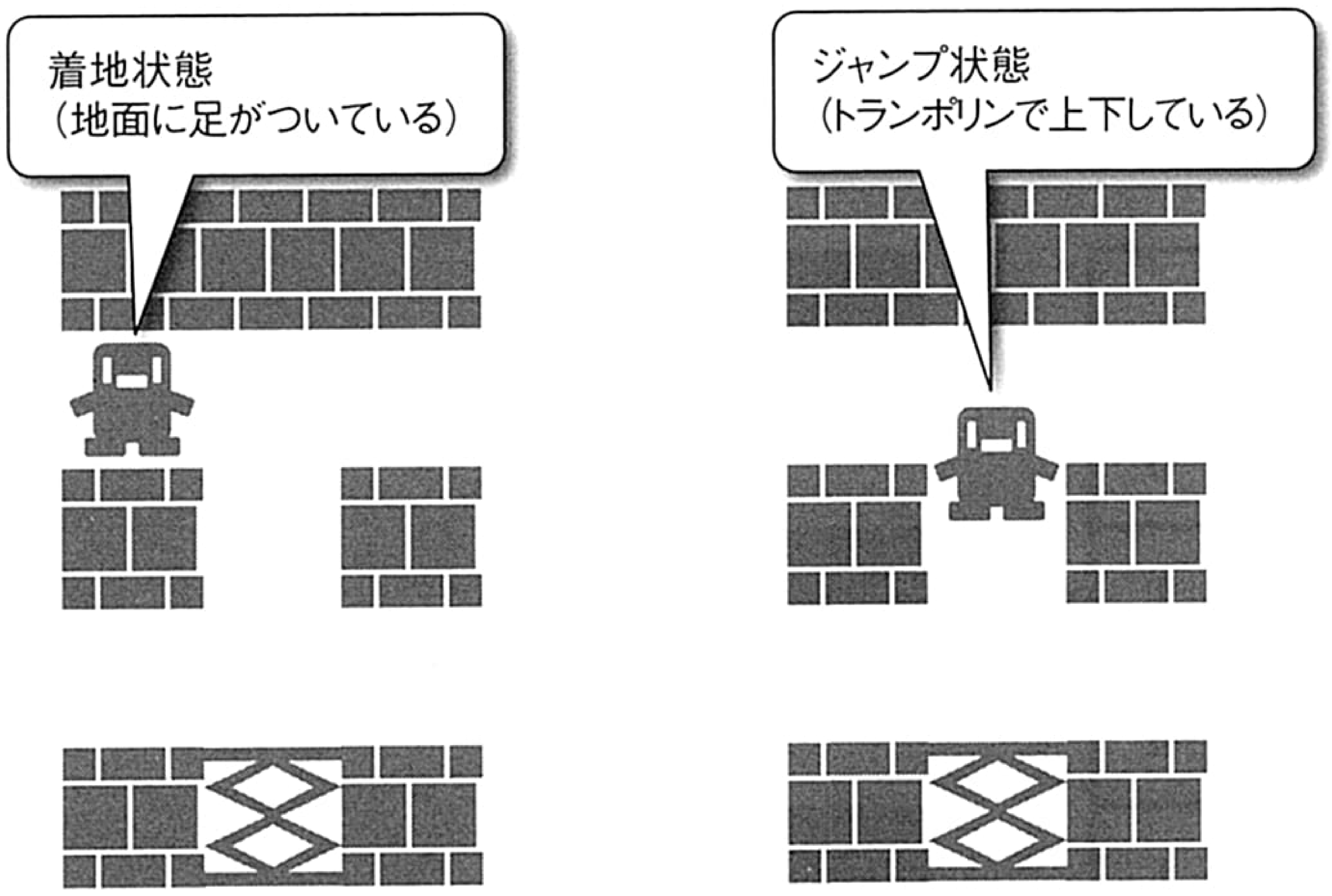


このサンプルでは同じトランポリンを何度も上下することができます。「マッピー」のように、同じトランポリンを連続して使っていると、トランポリンが破れるようにしてもよいでしょう。

トランポリンでジャンプする自機

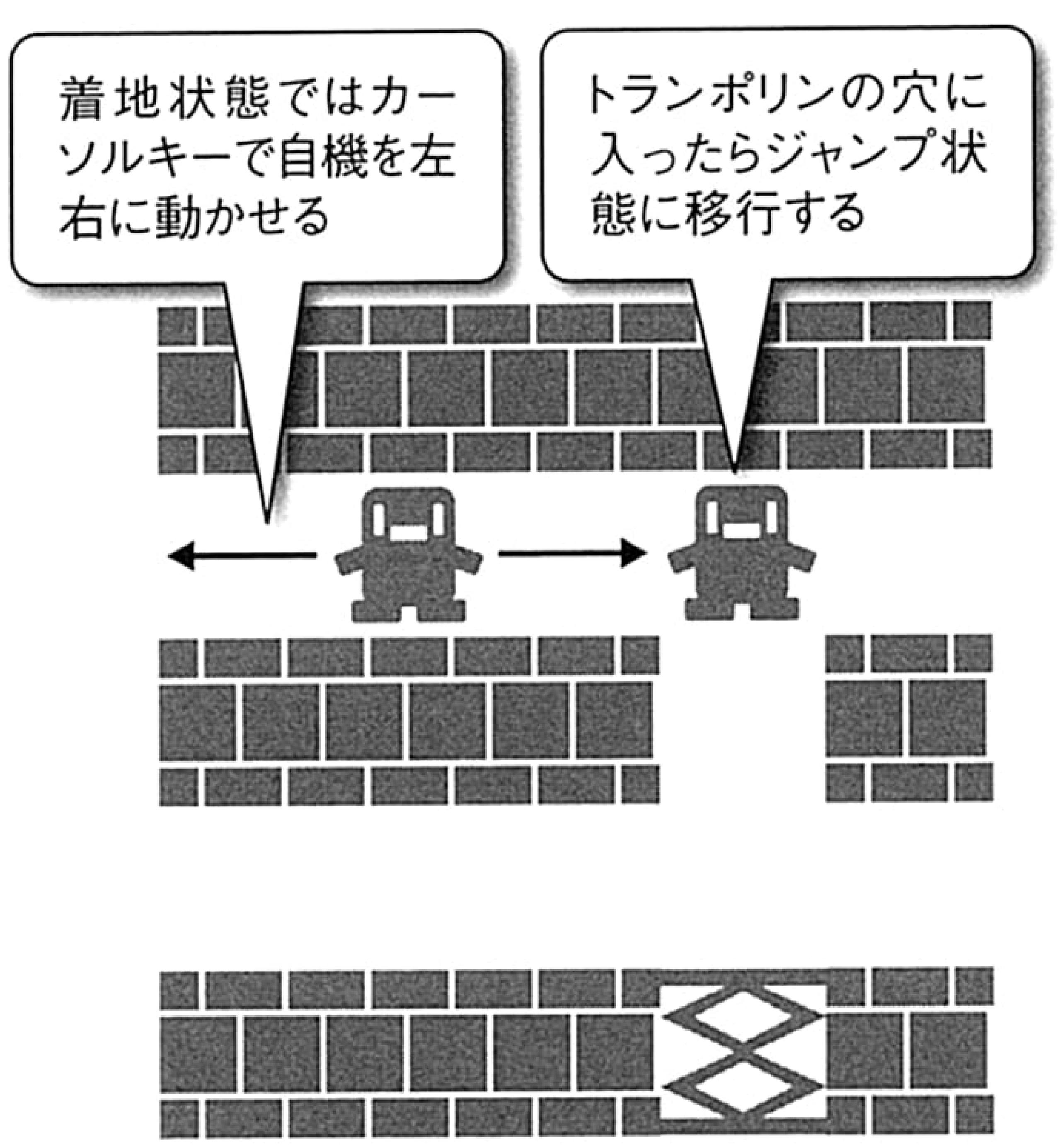
トランポリンでジャンプする自機には、着地状態とジャンプ状態という2つの状態を設けます(Fig.23)。着地状態は地面に足がついている状態で、ジャンプ状態はトランポリンを使って上下している状態です。

▼ **Fig.23**: 着地状態とジャンプ状態



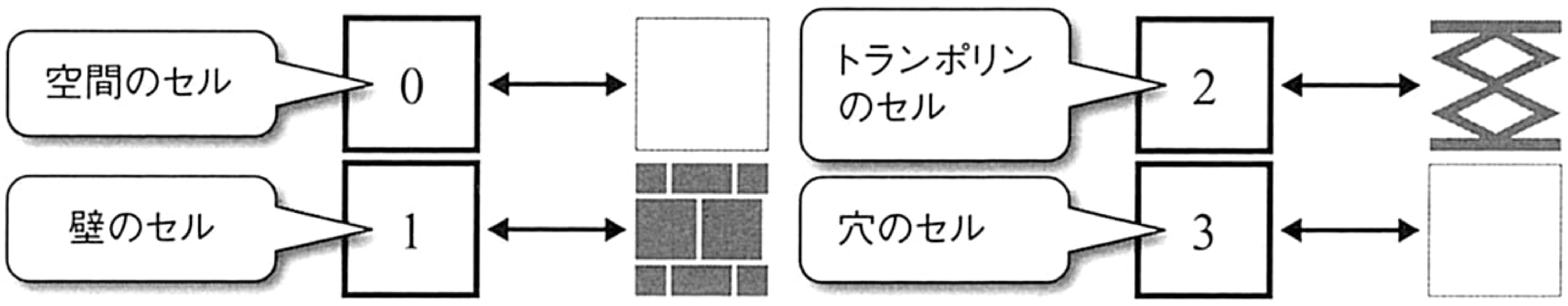
着地状態では、カーソルキーで自機を左右に動かすことができます(Fig.24)。自機がトランポリンの穴に入ったら、ジャンプ状態に移行します。

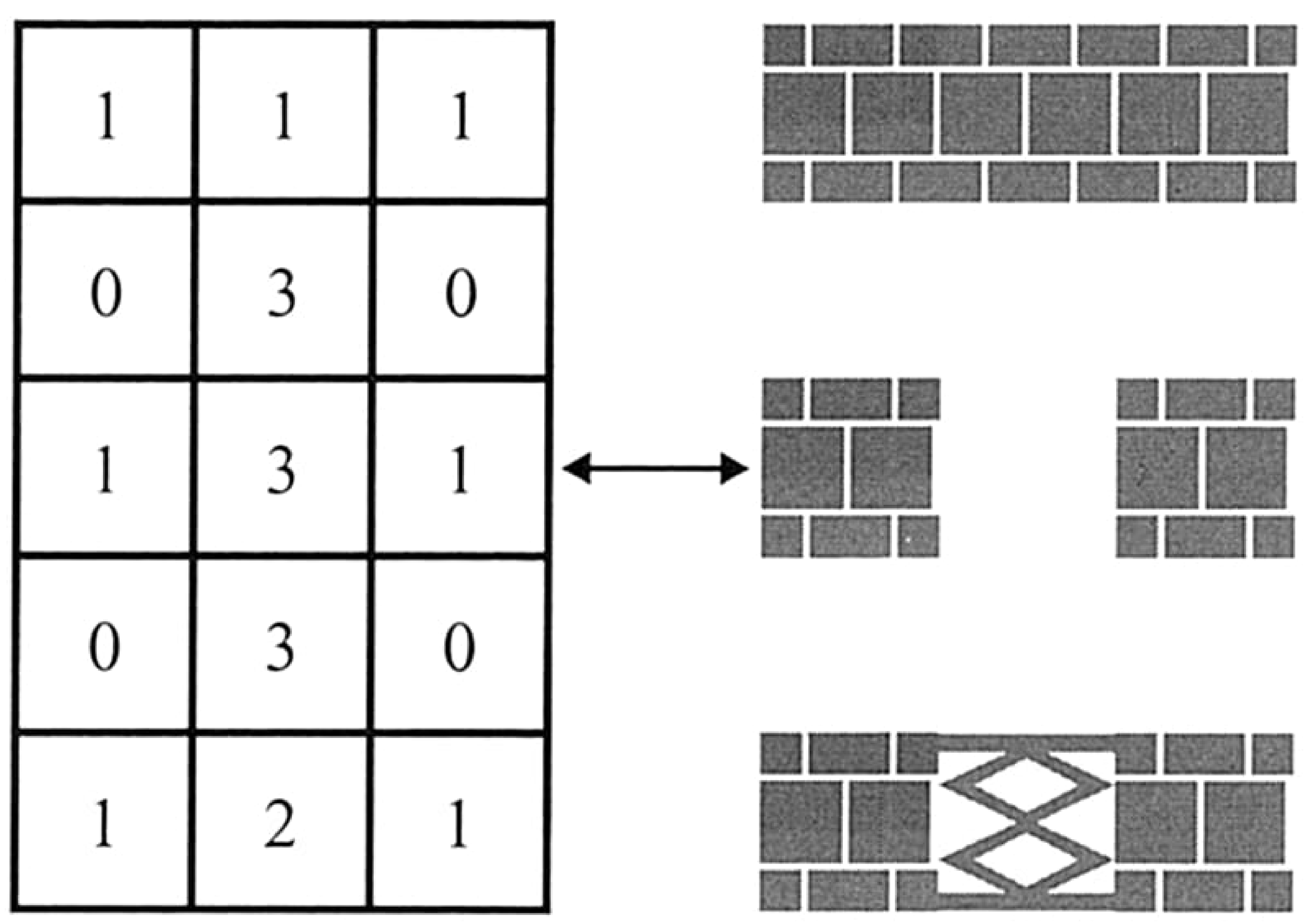
▼ **Fig.24**: 着地状態



自機が穴に入ったかどうかを判定するには、トランポリンの穴をセルで区別するとよいでしょう(Fig.25)。ここでは0～3の数値を使って、空間･壁･トランポリン･穴を表しました。自機がいるセルが穴だったら、自機が穴に入ったと判定します。

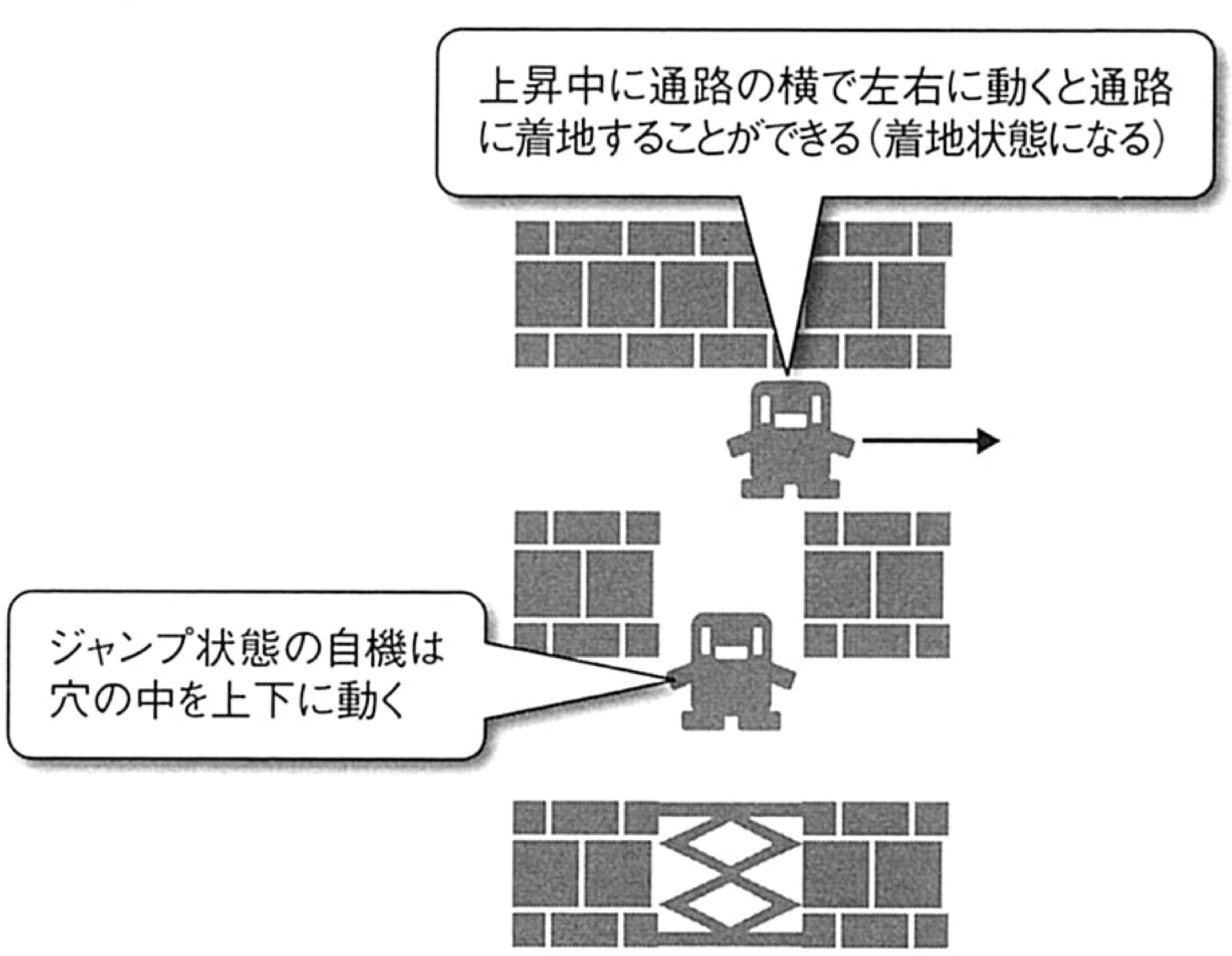
▼ **Fig.25**: トランポリンの穴





ジャンプ状態では、自機は穴の中を上下に動きます(Fig.26)。自機が上昇しているとき、通路の横でカーソルキーを左右に入力すると、通路に着地することができます。

▼ **Fig.26**: ジャンプ状態

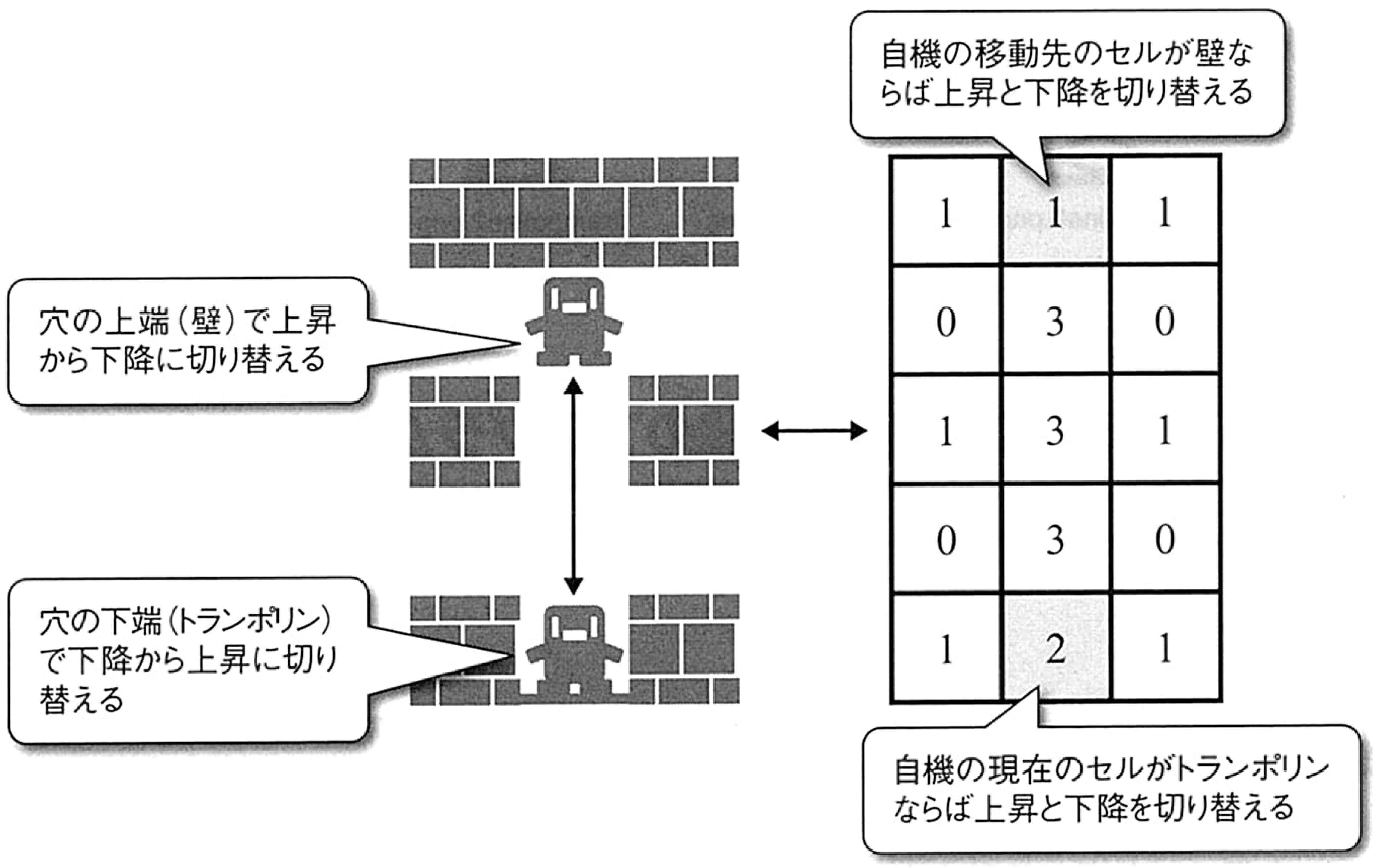


自機は穴の上端(壁)と、穴の下端(トランポリン)で、上昇と下降を切り替えます(Fig.27)。これは、

- 自機の移動先のセルが壁
- 自機の現在のセルがトランポリン

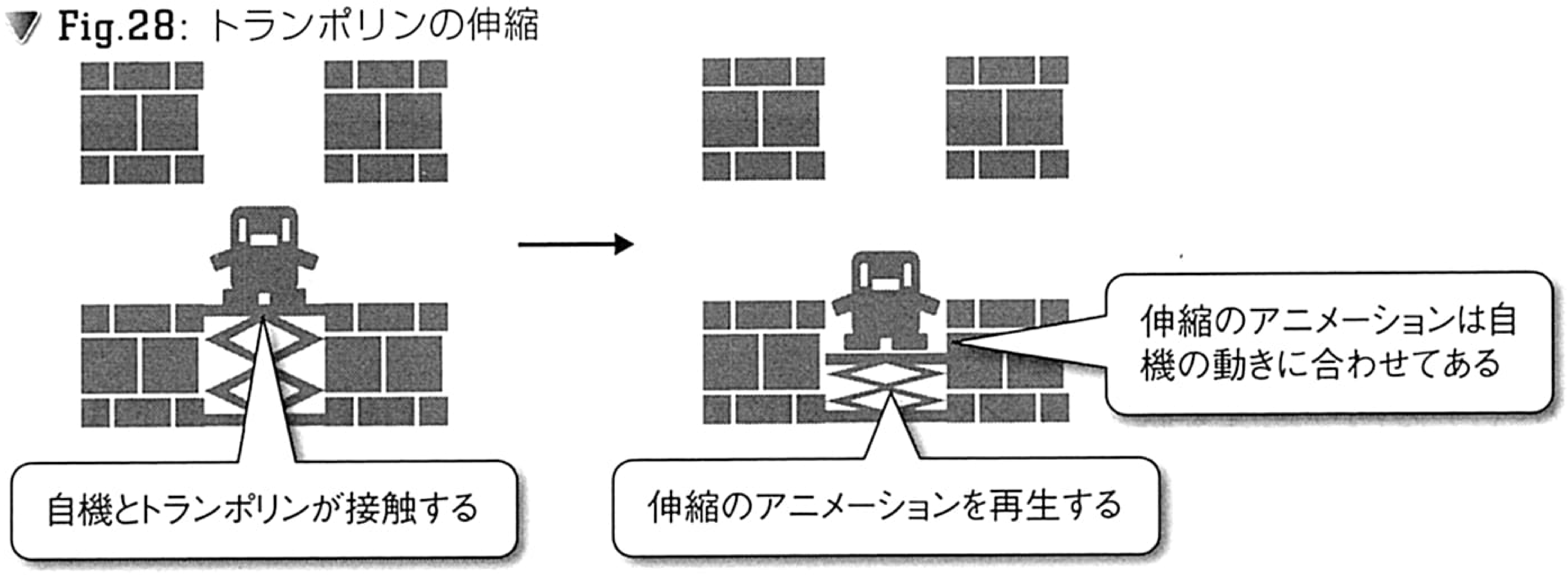
のいずれかの条件を満たしたときに、自機のY方向の速度を反転させれば実現できます。

▼ Fig.27: 上昇と下降の切り替え



自機がトランポリンに乗ったら、トランポリンは伸縮します(Fig.28)。自機とトランポリンの当たり判定処理を行い、接触したら、トランポリンに伸縮のアニメーションを設定します。

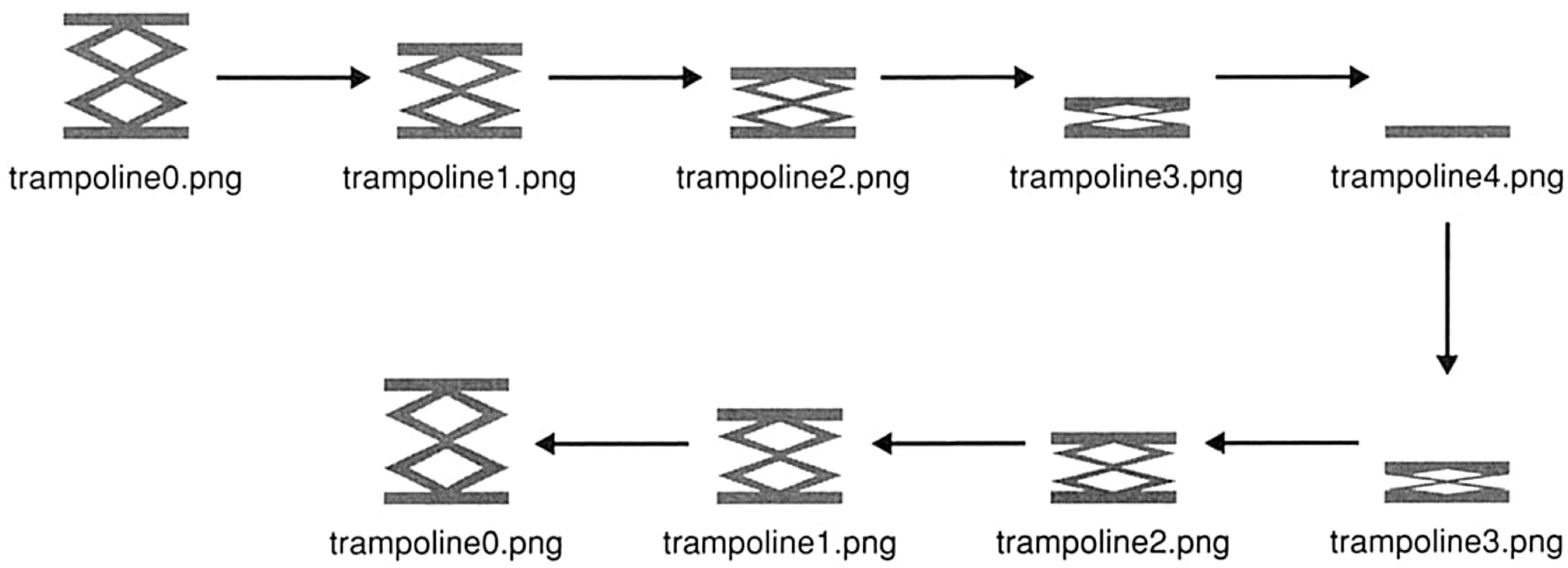
▼ Fig.28: トランポリンの伸縮



トランポリンが伸縮するアニメーションは、自機の動きに合わせてあります。自機が下降か

ら上昇に転ずるとき、自機の足下とトランポリンの板が同期して動くように、アニメーションを構成しています(Fig.29)。

▼ Fig.29: 伸縮のアニメーション



トランポリンでジャンプする自機のプログラム

List4は自機のプログラムです。自機の動きは着地状態とジャンプ状態に分かれています。着地状態では、

```
if (x==Math.floor(x)) {
```

のように、自機のX座標が整数かどうかを調べてから、

```
if (field.getCell(x, y)==CELL_PIT) {
```

のように、自機が現在いるセルが穴かどうかを判定します。穴ならば、

```
vx=0;
vy=1;
obj.jump=true;
```

のように、下方向の速度を設定し、ジャンプ状態にすることによって、自機に穴を落下させます。

ジャンプ状態では、

```
if (y==Math.floor(y)) {
```

のように、自機のY座標が整数かどうかを調べてから、

```
if (
    vy<0 &&
    field.getCell(x+vx, y)==CELL_VOID
) {
```

のように、自機が上昇しているかどうかと、自機の左右のセルが空間かどうかを判定します。条件を満たしていたら、

```
vy=0;
obj.jump=false;
```

のように、Y方向の速度を0にし、静止状態にすることによって、自機を通路に着地させます。
　続いて、

```
if (
    field.getCell(x, y)==CELL_TRAMPOLINE ||
    field.getCell(x, y+vy)==CELL_WALL
) {
```

のように、現在のセルがトランポリンかどうかと、移動先のセルが壁かどうかを調べます。いずれかの条件を満たしたら、

```
vy=-vy;
```

のように、Y方向の速度を反転することによって、上昇と下降を切り替えます。
　また、トランポリンを伸縮させるために、自機とトランポリンの当たり判定処理も行います。トランポリンに接触したら、

```
obj.sprite.onHitSprite=function(spr) {
    if (spr.getState()==STATE_TRAMPOLINE_STAY) {
        spr.initAnim(trampolineMoveAnim);
        spr.setState(STATE_TRAMPOLINE_MOVE);
    }
}
```

のようなonHitSprite関数を実行します。ここではトランポリンが静止している場合に、伸縮のアニメーションを設定し、伸縮状態に移行させます。

▼ **List4**: トランポリンでジャンプする自機(trampoline.js)

```
// グリッド(座標を補正するための定数)
var GRID=5;

// 自機のスピード
var SPEED=1.0/GRID;

// 自機
function Player(x, y) {
  var obj=this;

  // スプライトの作成
  obj.sprite=game.createSprite();

  // アニメーションとグリッドの設定
  obj.sprite.initAnim(playerWalkAnim);
  obj.sprite.setGrid(GRID);

  // 座標と速度の設定
  obj.sprite.setXY(x, y);
  obj.vy=0;

  // ジャンプ状態(true)か着地状態(false)か
  obj.jump=false;

  // 更新処理
  obj.sprite.onUpdate=function() {

    // 座標と速度の取得
    var x=obj.sprite.getX(), y=obj.sprite.getY();
    var vx, vy=obj.vy;

    // カーソルキーの入力に応じて左右に移動する
    vx=0;
    if (game.isKey(KEY_LEFT)) vx=-1;
    if (game.isKey(KEY_RIGHT)) vx=1;

    // 着地状態
    if (!obj.jump) {

      // 自機のX座標が整数のとき
      if (x==Math.floor(x)) {

        // 移動方向のセルが壁ならば、移動しない
```

```
      if (field.getCell(x+vx, y)==CELL_WALL) {
        vx=0;
      }

      // 現在のセルが穴ならば、落下する
      if (field.getCell(x, y)==CELL_PIT) {

        // 下方向の速度を設定する
        vx=0;
        vy=1;

        // ジャンプ状態にする
        obj.jump=true;
      }
    }

    // 左右への移動中はアニメーションを再生する
    if (vx!=0) {
      obj.sprite.stepAnim();
    }
  } else

  // ジャンプ状態
  {
    // 自機のY座標が整数のとき
    if (y==Math.floor(y)) {

      // 自機が上昇中で、
      // 左右への移動先のセルが空間ならば、着地する
      if (
        vy<0 &&
        field.getCell(x+vx, y)==CELL_VOID
      ) {
        // Y方向の速度を0にする
        vy=0;

        // 着地状態にする
        obj.jump=false;
      } else {

        // 着地するとき以外は、左右には移動しない
        vx=0;
      }

      // 現在のセルがトランポリンか、
      // 移動先のセルが壁ならば、
      // 上昇と下降を切り替える
      if (
```

```
          field.getCell(x, y)==CELL_TRAMPOLINE ||
          field.getCell(x, y+vy)==CELL_WALL
        ) {
          // Y方向の速度を反転する
          vy=-vy;
        }
      } else {

        // 自機のY座標が整数でないときには、左右には移動しない
        vx=0;
      }

      // トランポリンに接触したときの処理
      obj.sprite.onHitSprite=function(spr) {

        // トランポリンが静止状態ならば、
        // トランポリンが伸縮するアニメーションを設定し、
        // 伸縮状態にする
        if (spr.getState()==STATE_TRAMPOLINE_STAY) {
          spr.initAnim(trampolineMoveAnim);
          spr.setState(STATE_TRAMPOLINE_MOVE);
        }
      }

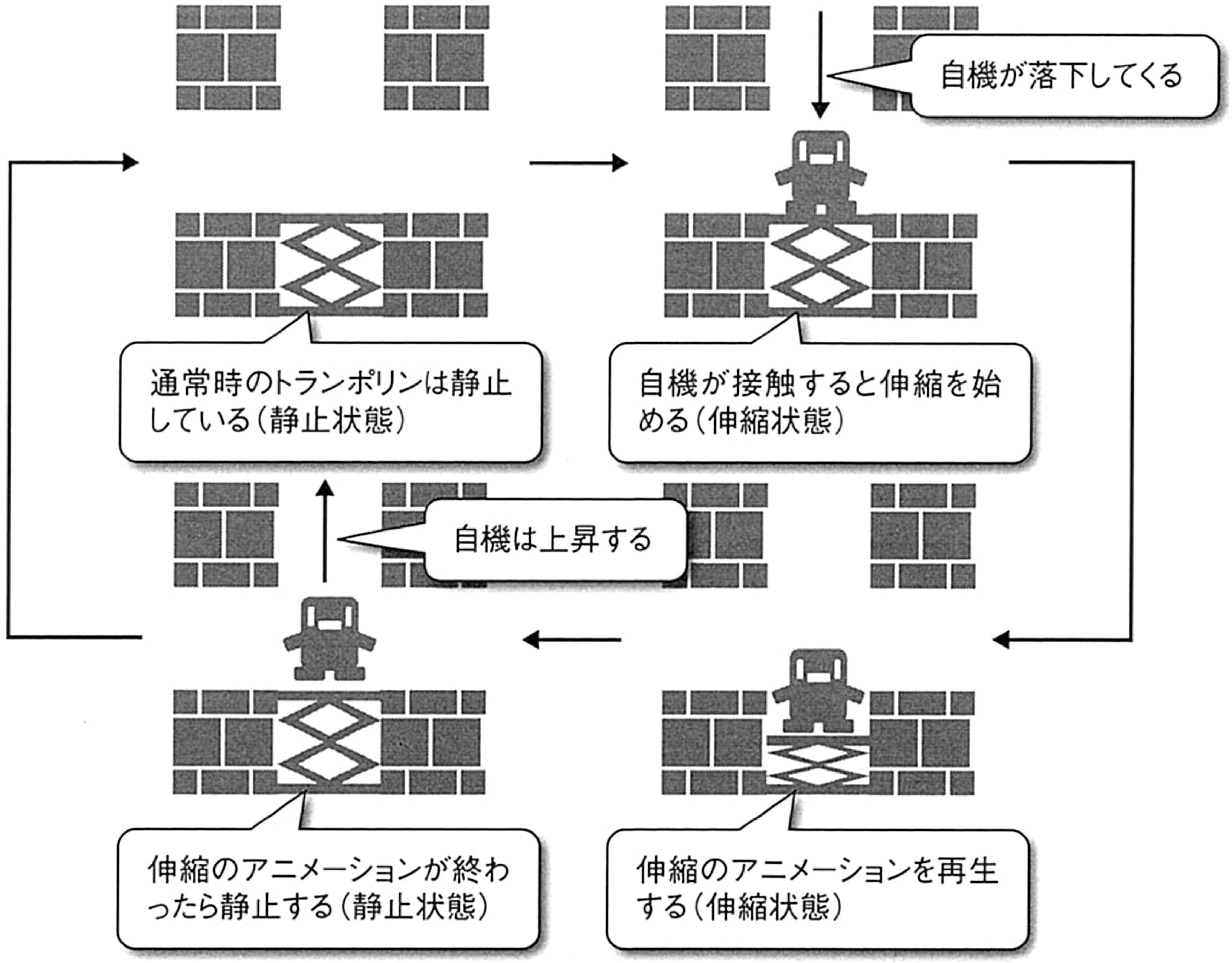
      // 当たり判定処理
      game.testHitSprite(obj.sprite);
    }

    // 座標と速度の更新
    obj.sprite.setXY(x+vx*SPEED, y+vy*SPEED);
    obj.vy=vy;
  }
}
```

伸縮するトランポリン

トランポリンは通常は静止していますが、自機が接触すると伸縮します。伸縮が終わると、再び静止します。ここでは静止しているときを静止状態、伸縮しているときを伸縮状態と呼ぶことにします(Fig.30)。

▼ Fig.30: トランポリンの状態



List5はトランポリンのプログラムです。トランポリンの処理はTrampolineオブジェクトにまとめました。

最初は、

```
obj.sprite.setState(STATE_TRAMPOLINE_STAY);
```

のように、トランポリンを静止状態にします。静止状態のトランポリンに自機が接触すると、トランポリンは伸縮のアニメーションを開始し、伸縮状態になります(p.279)。

伸縮のアニメーションが終わったら、

```
if (obj.sprite.isAnimOver()) {
    obj.sprite.initAnim(trampolineStayAnim);
    obj.sprite.setState(STATE_TRAMPOLINE_STAY);
}
```

のように、静止のアニメーションを設定し、静止状態に戻ります。静止のアニメーションは、アニメーションといっても動きはなく、同じ画像のまま止まっています。

▼ **List5**: 伸縮するトランポリン(trampoline.js)

```
// 荷物の状態(静止・伸縮)
var STATE_TRAMPOLINE_STAY=0;
var STATE_TRAMPOLINE_MOVE=1;

// トランポリン
function Trampoline(x, y) {
  var obj=this;

  // スプライトの作成
  obj.sprite=game.createSprite();

  // アニメーション・グリッド・座標の設定
  obj.sprite.initAnim(trampolineStayAnim);
  obj.sprite.setGrid(GRID);
  obj.sprite.setXY(x, y);

  // 最初は静止状態にする
  obj.sprite.setState(STATE_TRAMPOLINE_STAY);

  // 更新処理
  obj.sprite.onUpdate=function() {

    // アニメーション(伸縮のアニメーション)が終わったら、
    // 静止のアニメーションを設定し、静止状態に移行する
    if (obj.sprite.isAnimOver()) {
      obj.sprite.initAnim(trampolineStayAnim);
      obj.sprite.setState(STATE_TRAMPOLINE_STAY);
    }

    // アニメーションを再生する
    obj.sprite.stepAnim();
  }
}
```

トランポリンでジャンプするゲームの本体

List6はトランポリンのゲームの本体です。他のサンプルと同様に、初期化処理(init関数)とメインルーチン(main関数)があります。

初期化処理では、

```
field.initCellImage(new Array(
    "void.png", "wall.png", "void.png",
    "void.png", "void.png"
));
```

のように、セルの種類に対応する画像の一覧を設定します。ここでは、壁のセルには壁の画像(wall.png)を、空間・トランポリン・穴・自機のセルには空白の画像(void.png)を割り当てています。トランポリンや自機を空白にするのは、これらはスプライトとして表示するからです。他のサンプルでも、スプライトとして表示するセルには空白の画像(void.png)を割り当てています。

List6: トランポリンでジャンプするゲームの本体(trampoline.js)

```
// セルの種類(空間・壁・トランポリン・穴・自機)
var CELL_VOID=0;
var CELL_WALL=1;
var CELL_TRAMPOLINE=2;
var CELL_PIT=3;
var CELL_PLAYER=4;

// ゲーム本体・自機・トランポリン・フィールドのオブジェクト
var game;
var player;
var trampoline;
var field;

// 自機とトランポリンのアニメーション
var playerWalkAnim;
var trampolineStayAnim;
var trampolineMoveAnim;

// 初期化
function init() {
  var x, y, i;

  // 画面の消去
  game.clear();

  // フィールドの作成
  field=game.createField();

  // セルの種類に対応する画像の一覧を配列で設定する
  field.initCellImage(new Array(
    "void.png", "wall.png", "void.png",
```

```
    "void.png", "void.png"
  ));

  // セルの初期化
  field.initCell(new Array(
    "111111111111",
    "134030030031",
    "131131131131",
    "130030030031",
    "131121121131",
    "130000000031",
    "121111111121"
  ));

  // 自機とトランポリンを生成する
  trampoline=new Array();
  for (i=0, y=0; y<field.getYCount(); y++) {
    for (x=0; x<field.getXCount(); x++) {
      switch (field.getCell(x, y)) {

        // 自機の生成
        case CELL_PLAYER:
          player=new Player(x, y);
          field.setCell(x, y, CELL_VOID);
          break;

        // トランポリンの生成
        case CELL_TRAMPOLINE:
          trampoline[i++]=new Trampoline(x, y);
          break;
      }
    }
  }
}

// メインルーチン
function main() {

  // ゲーム本体の作成
  game=new Game();

  // アニメーションの作成
  playerWalkAnim=new Array(
    "pcWalk0.png", "pcWalk2.png", "pcWalk0.png",
    "pcWalk1.png", "pcWalk3.png", "pcWalk1.png"
  );
  trampolineStayAnim=new Array(
```

```
    "trampoline0.png"
  );
  trampolineMoveAnim=new Array(
    "trampoline0.png", "trampoline1.png",
    "trampoline2.png", "trampoline3.png",
    "trampoline4.png", "trampoline4.png",
    "trampoline3.png", "trampoline2.png",
    "trampoline1.png", "trampoline0.png"
  );

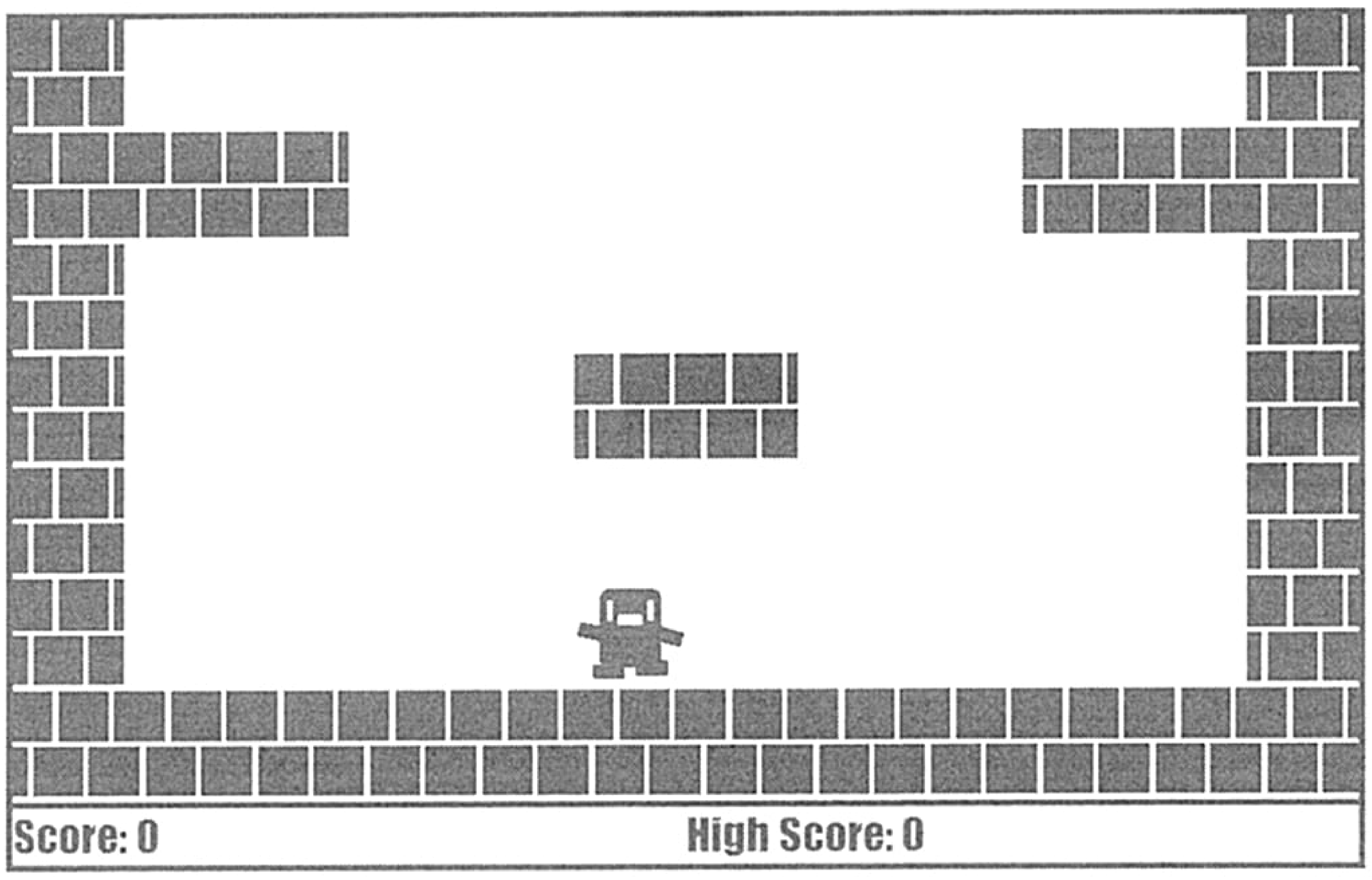
  // ゲームの初期化
  init();
}
```

床を押し上げる

続いて紹介するのは、ジャンプを使った色々なアクションです。ジャンプして床に登ったり、床から飛び降りたりします。また、床に下からぶつかることによって、床を押し上げるアクションも行います。こういった動きは「マリオブラザーズ」(任天堂 1983年)や「スーパーマリオブラザーズ」(任天堂 1985年)あるいは「アイスクライマー」(任天堂 1985年)などをはじめとして、さまざまなゲームに採用されています。

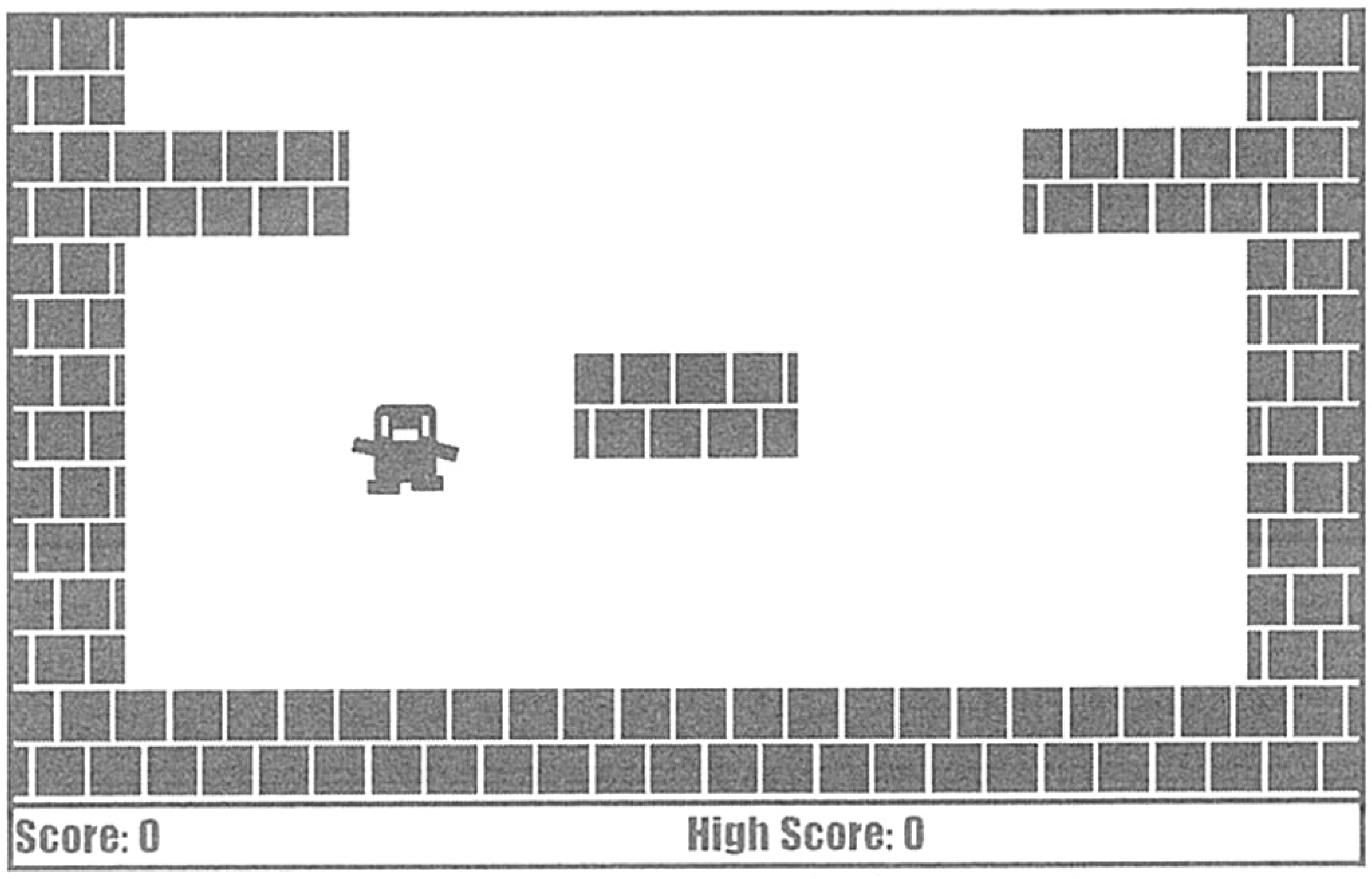
床を押し上げるサンプルを実行するには、liftFloor.htmlをブラウザで開きます(Fig.31)。自機は画面下方に表示されます。

▼ **Fig.31**: 床を押し上げるサンプル



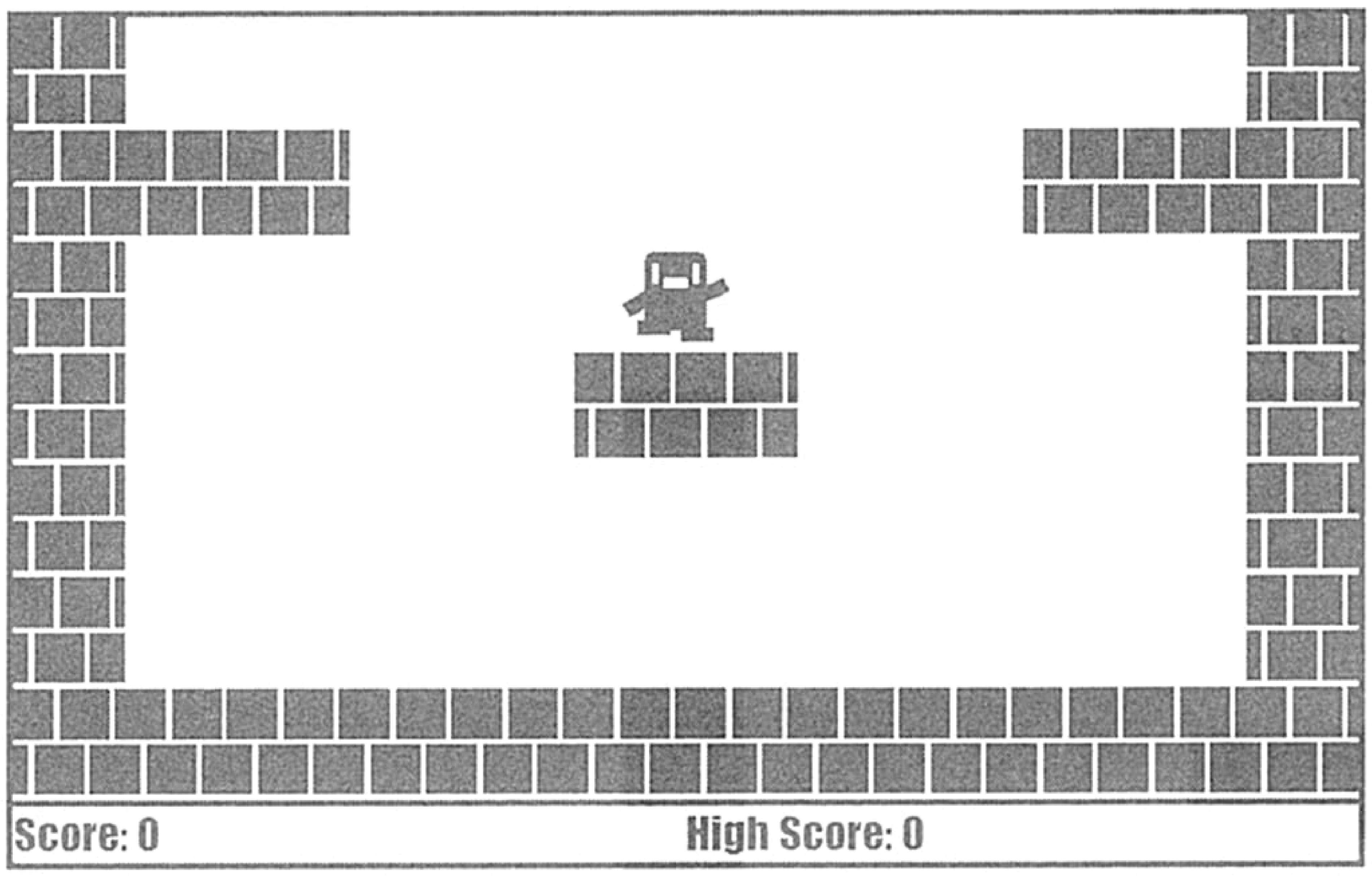
自機はカーソルキーで左右に動かすことができます。また、スペースキーを押すとジャンプします(Fig.32)。

▼ **Fig.32**: ジャンプする自機

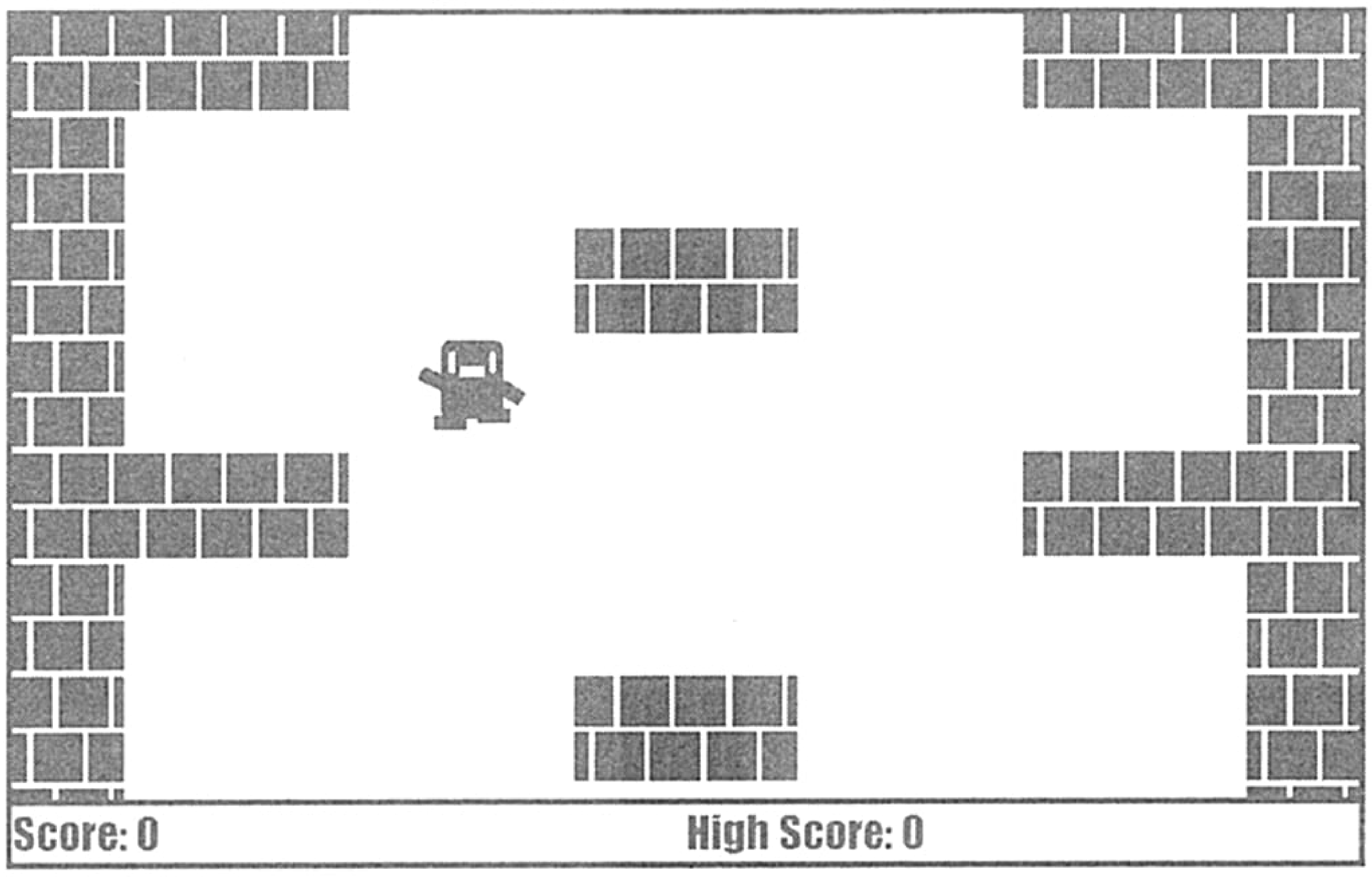


上手にジャンプすると、高いところにある床に着地することができます(Fig.33)。その床でさらにジャンプすると、もっと高いところに登ることができます(Fig.34)。自機の動きに合わせて、フィールドが上下にスクロールします。

▼ **Fig.33**: 床に着地する



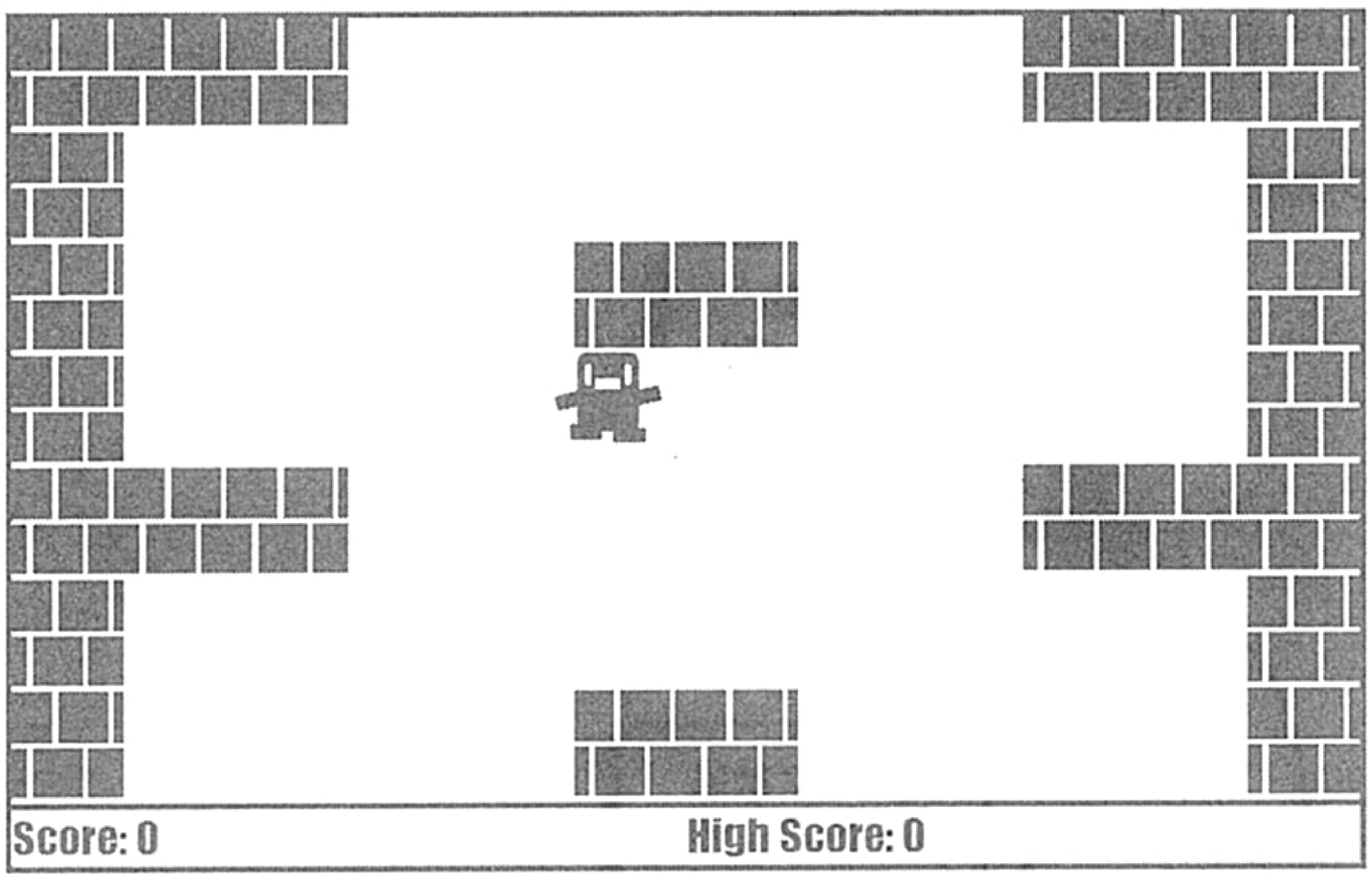
▼ **Fig.34**: 床の上でさらにジャンプする



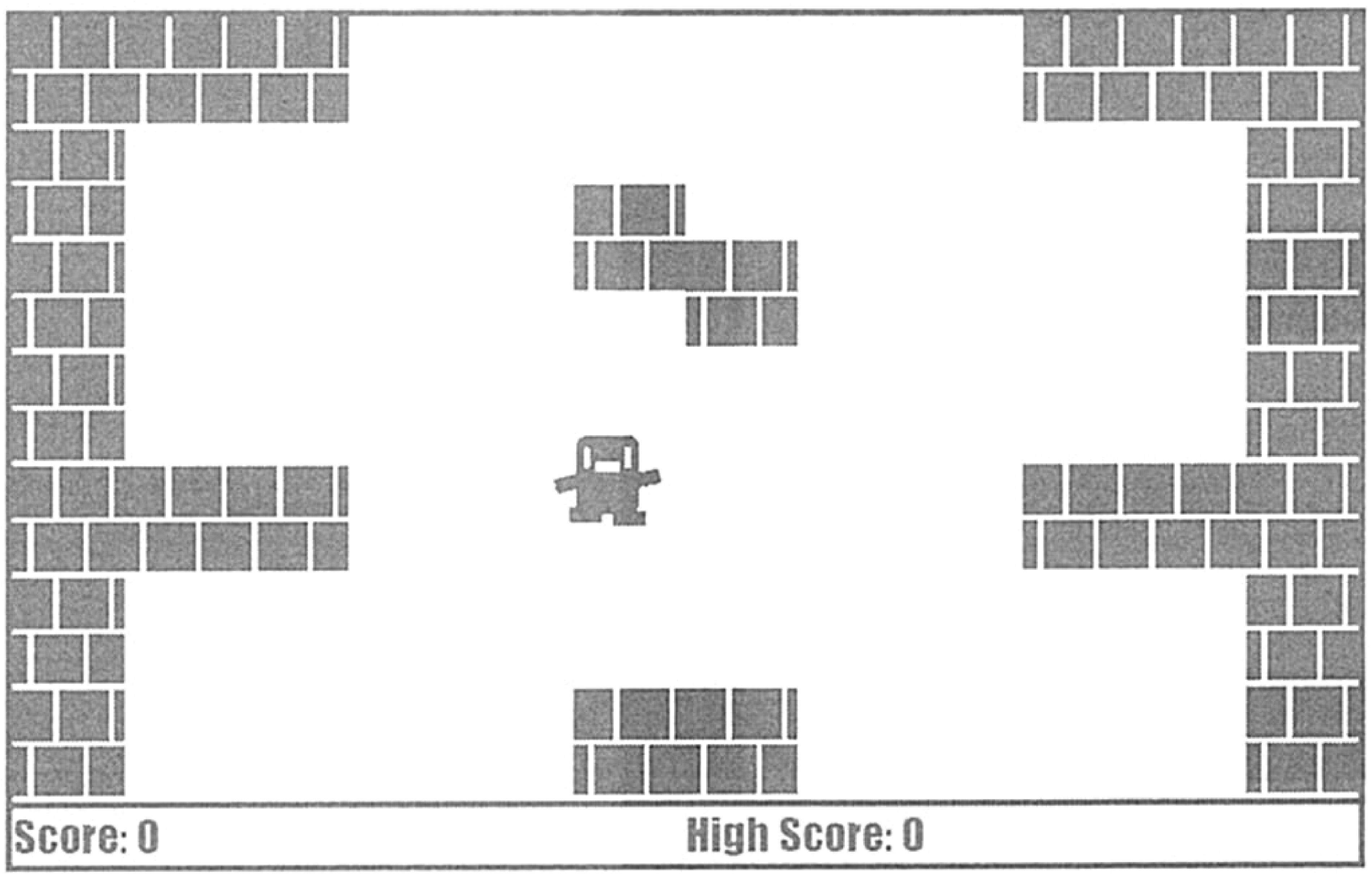
床から落ちると自機は落下します。自機が落下するときも、自機の動きに合わせてフィールドがスクロールします。

ジャンプしたときに床に下からぶつかると、床を押し上げることができます(Fig.35)。押し上げられた床は少し上昇してから下降し、一定時間後には元の位置に戻ります(Fig.36)。床にぶつかった自機は落下します。

▼ **Fig.35**: 床に下からぶつかる



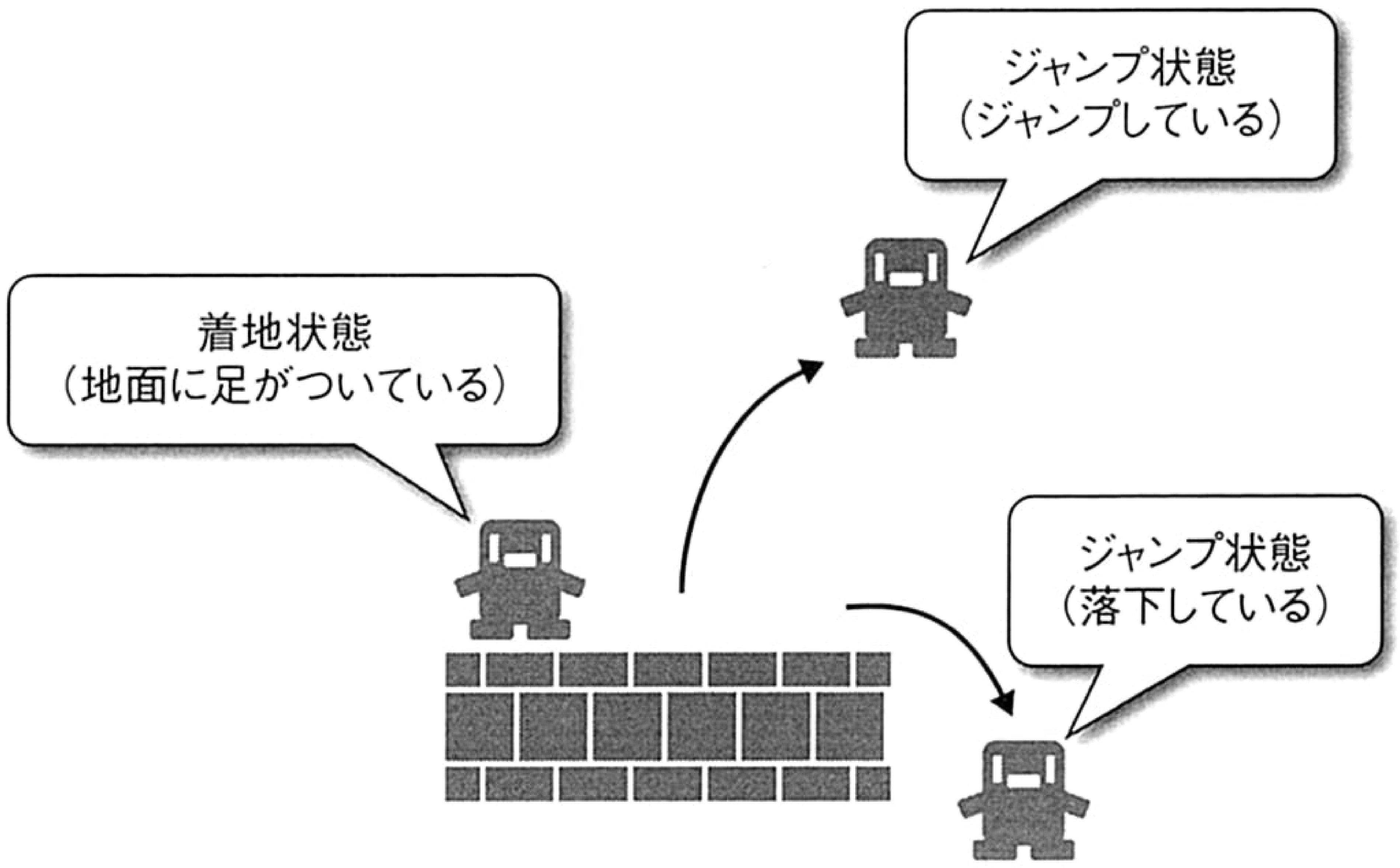
▼ Fig.36: 床が上昇する



床を押し上げる自機

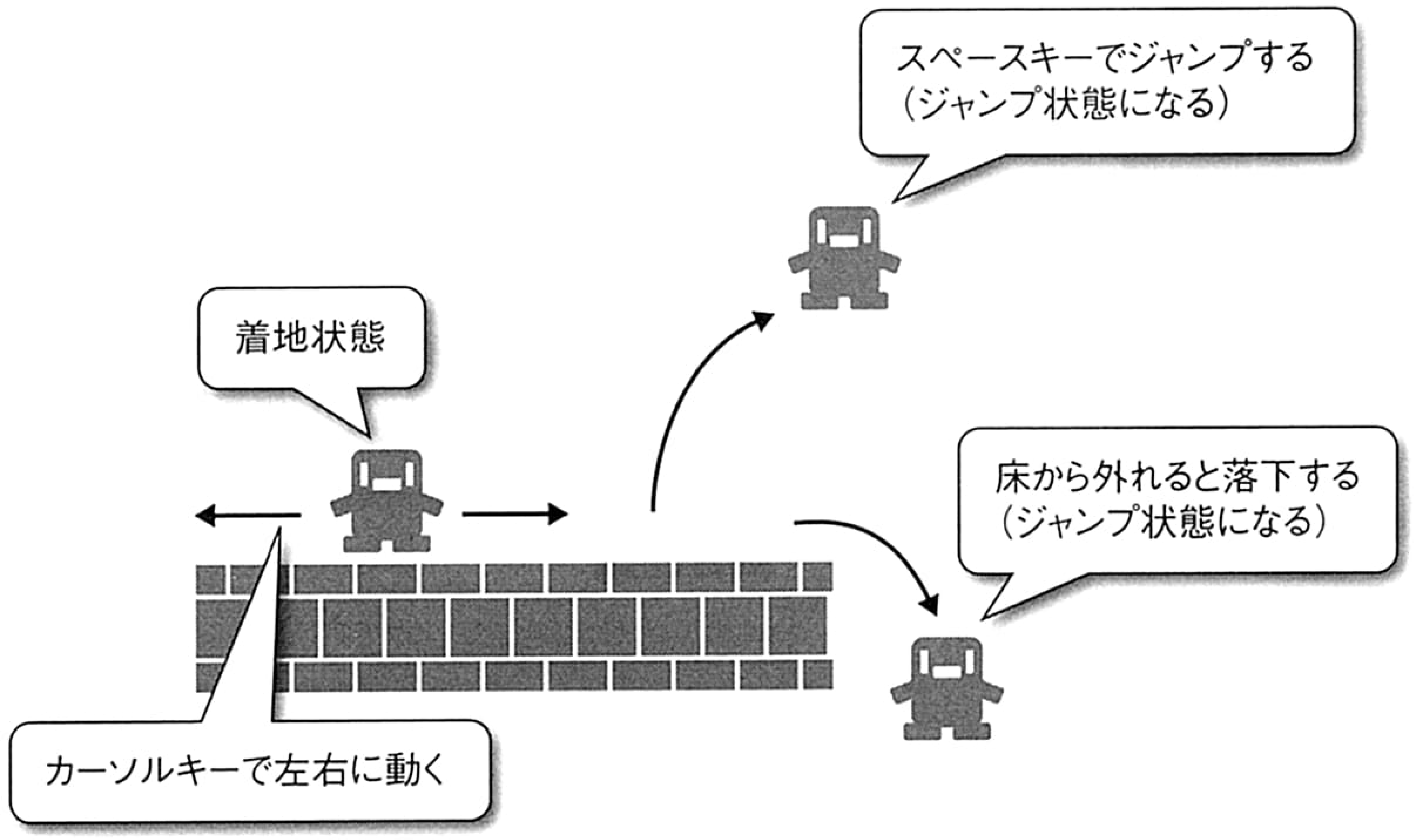
トランポリンでジャンプする自機(p.275)と同様に、床を押し上げる自機には、着地状態とジャンプ状態という2つの状態を設けます(Fig.37)。着地状態は地面に足がついている状態で、ジャンプ状態はジャンプまたは落下している状態です。

▼ Fig.37: 着地状態とジャンプ状態



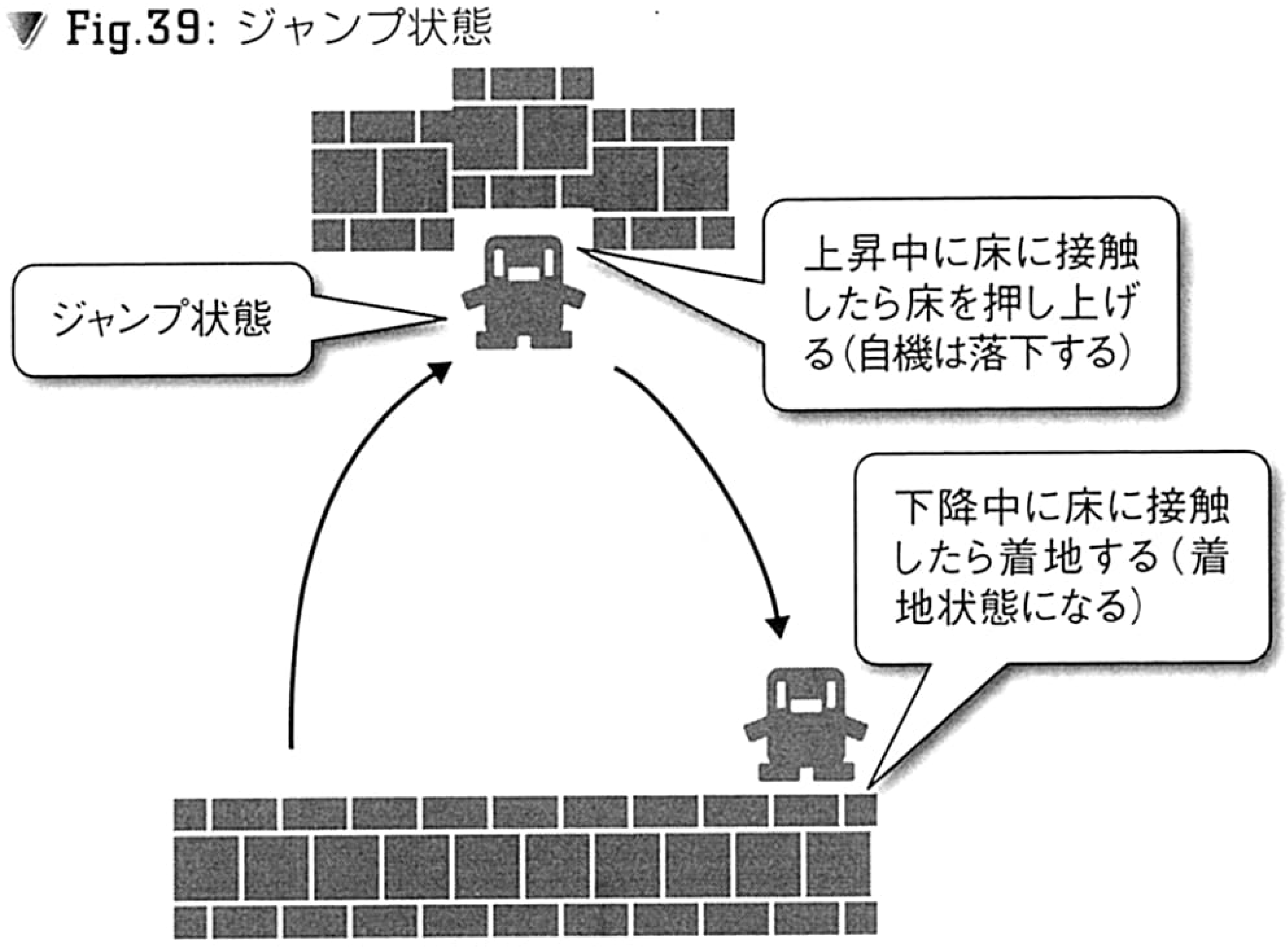
着地状態ではカーソルキーで自機を左右に動かすことができます(Fig.38)。スペースキーを押してジャンプするか、床から外れて落下したら、ジャンプ状態に移行します。

▼ Fig.38：着地状態



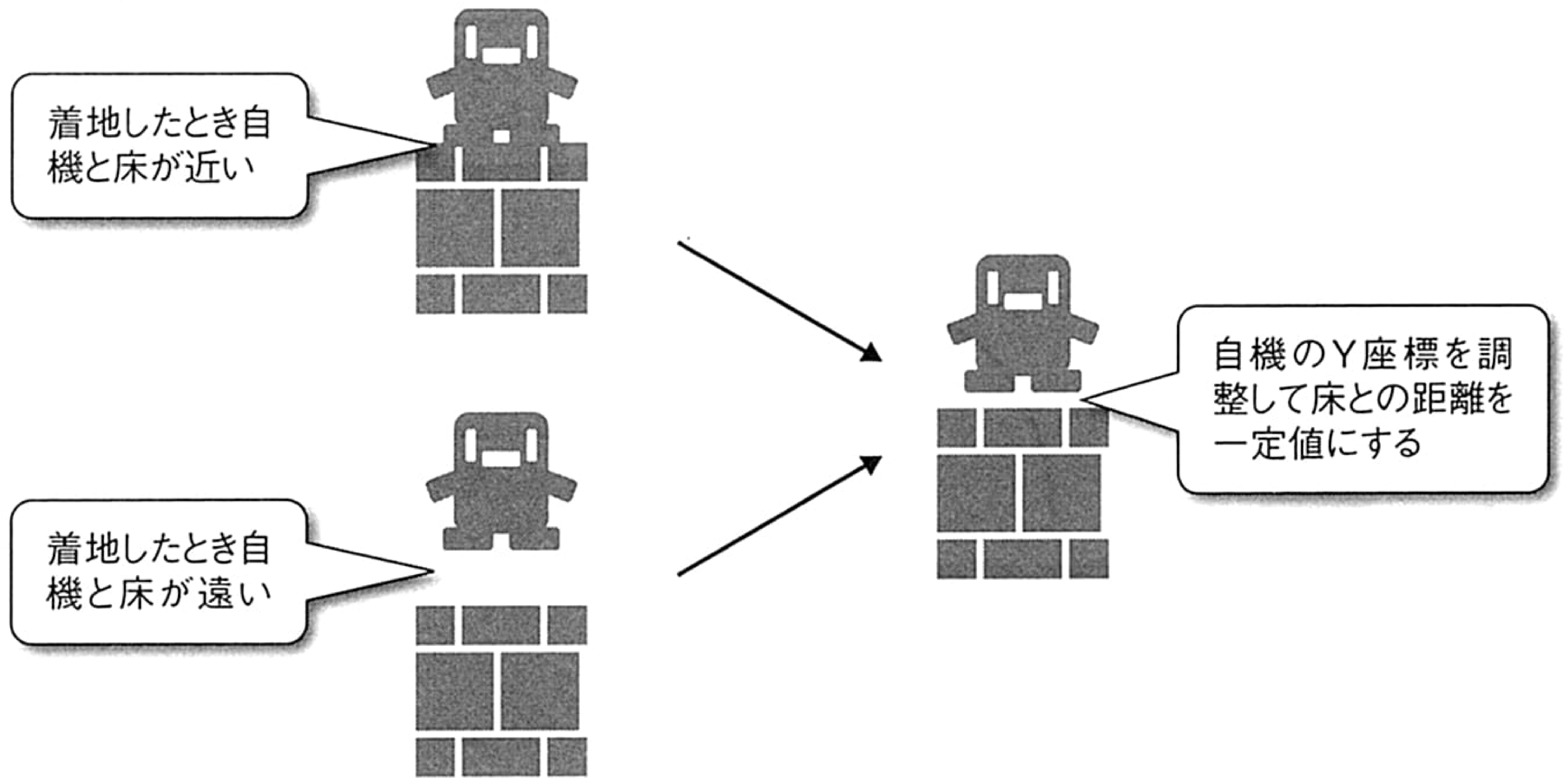
ジャンプ状態で上昇しているときに床に接触したら、床を押し上げます(Fig.39)。下降しているときに床に接触したら、着地します。

▼ Fig.39：ジャンプ状態



床に着地したら、着地状態に戻ります。このとき、自機が床の上にちょうどよく乗るように、自機のY座標を調整します(Fig.40)。この調整が必要なのは、着地するときの自機の座標は場合によって多少ずれるからです。

▼ **Fig.40**: 着地時にY座標を調整する



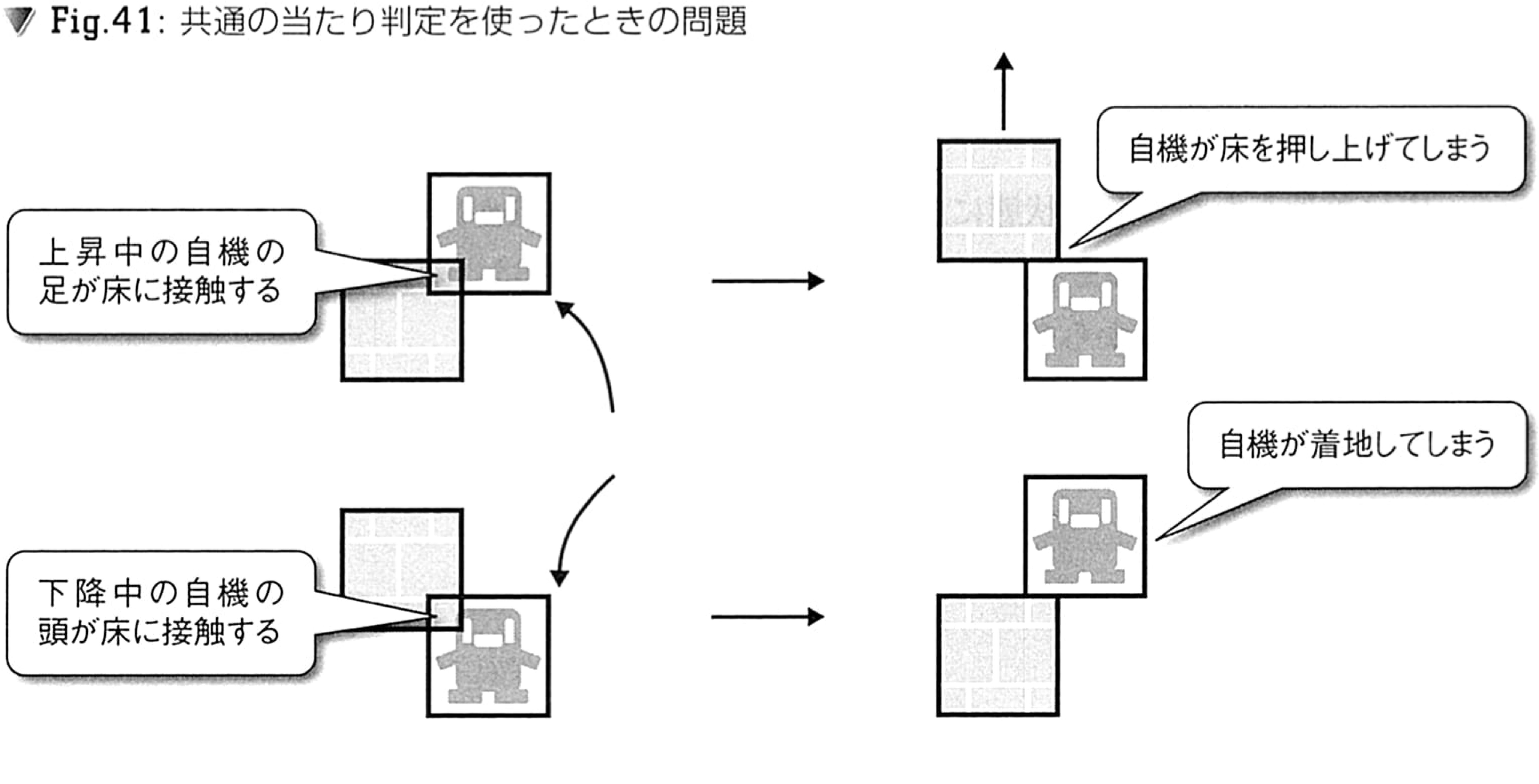
当たり判定の使い分け

自機と床(または壁)の間には、次のような関係があります。

(1) 自機が左右に動くときに壁にぶつかったら、移動しない
(2) 自機が上昇しているときに床に接触したら、床を押し上げる
(3) 自機が下降しているときに床に接触したら、着地する

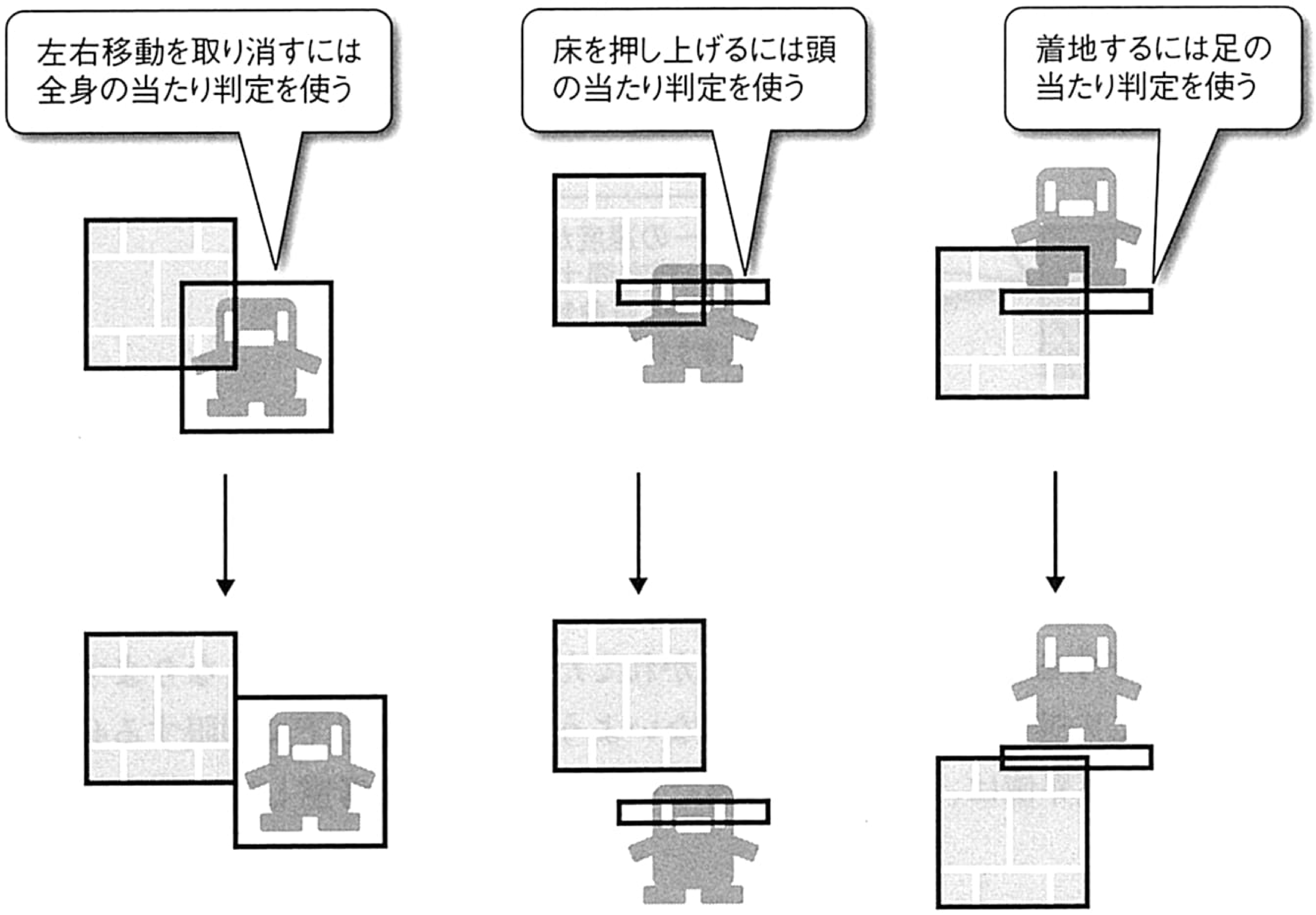
これらのアクションは自機と床の当たり判定処理を使って実現しますが、ここで少し工夫が必要です。どのアクションでも自機の当たり判定を共通にすると、動きが不自然になることがあります(Fig.41)。足が床にぶつかったのに床を押し上げてしまったり、頭がぶつかったのに床に着地してしまったりといったことが起こります。

▼ **Fig.41**: 共通の当たり判定を使ったときの問題



この問題を回避する方法の一つは、アクションによって異なる当たり判定を使うことです(Fig.42)。サンプルでは、(1)には全身の当たり判定、(2)には頭の当たり判定、(3)には足の当たり判定を使います。当たり判定の位置や大きさを調節して、なるべく動きが自然に見えるようにします。

▼ **Fig.42**: 異なる当たり判定を使い分ける

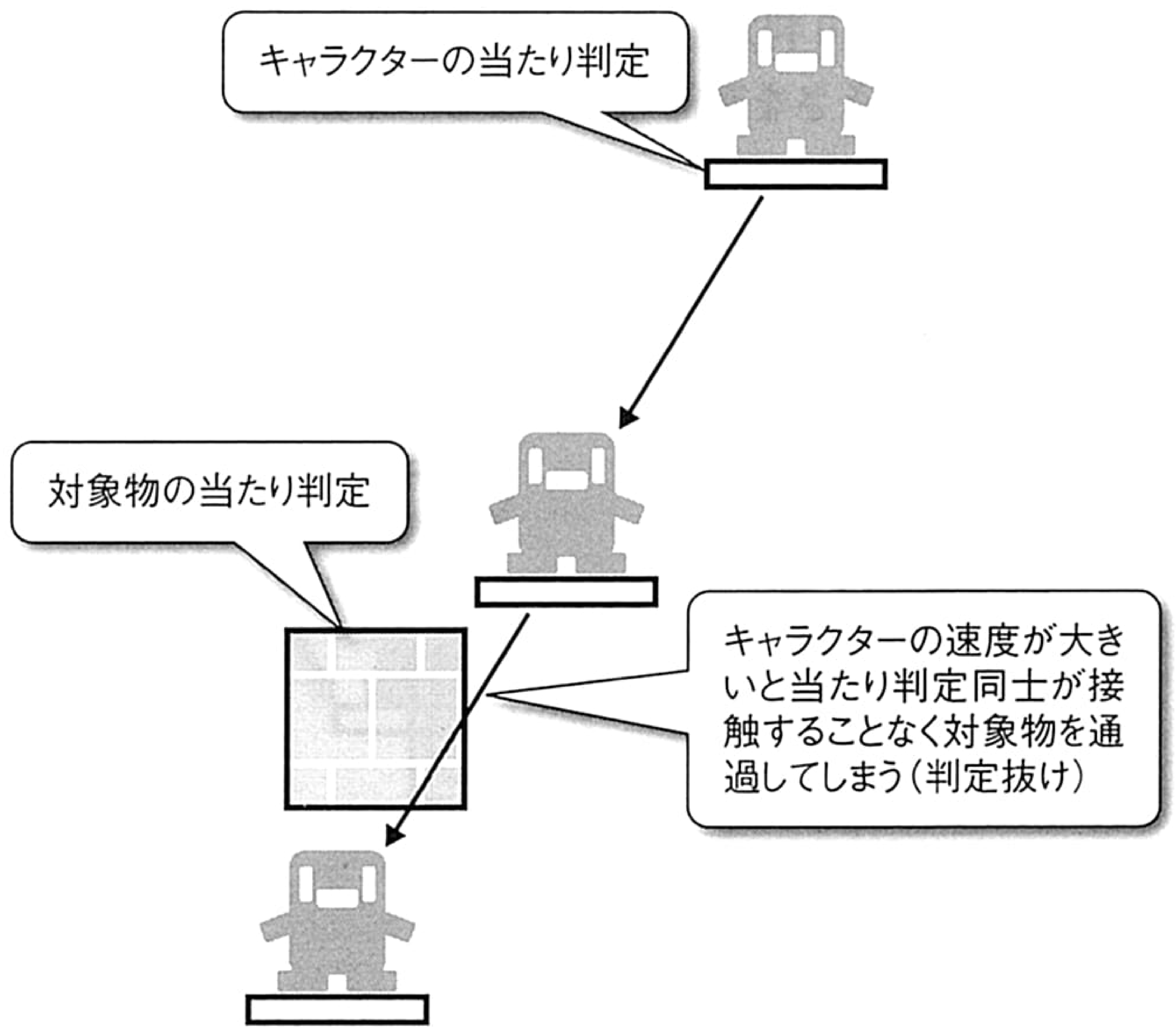


判定抜けの防止

判定抜けというのは、当たり判定処理における問題の一つです。当たり判定処理が不適切なために、例えばキャラクターが本来通過できない壁や床などを通り抜けてしまうようなことを、判定抜けと呼びます。

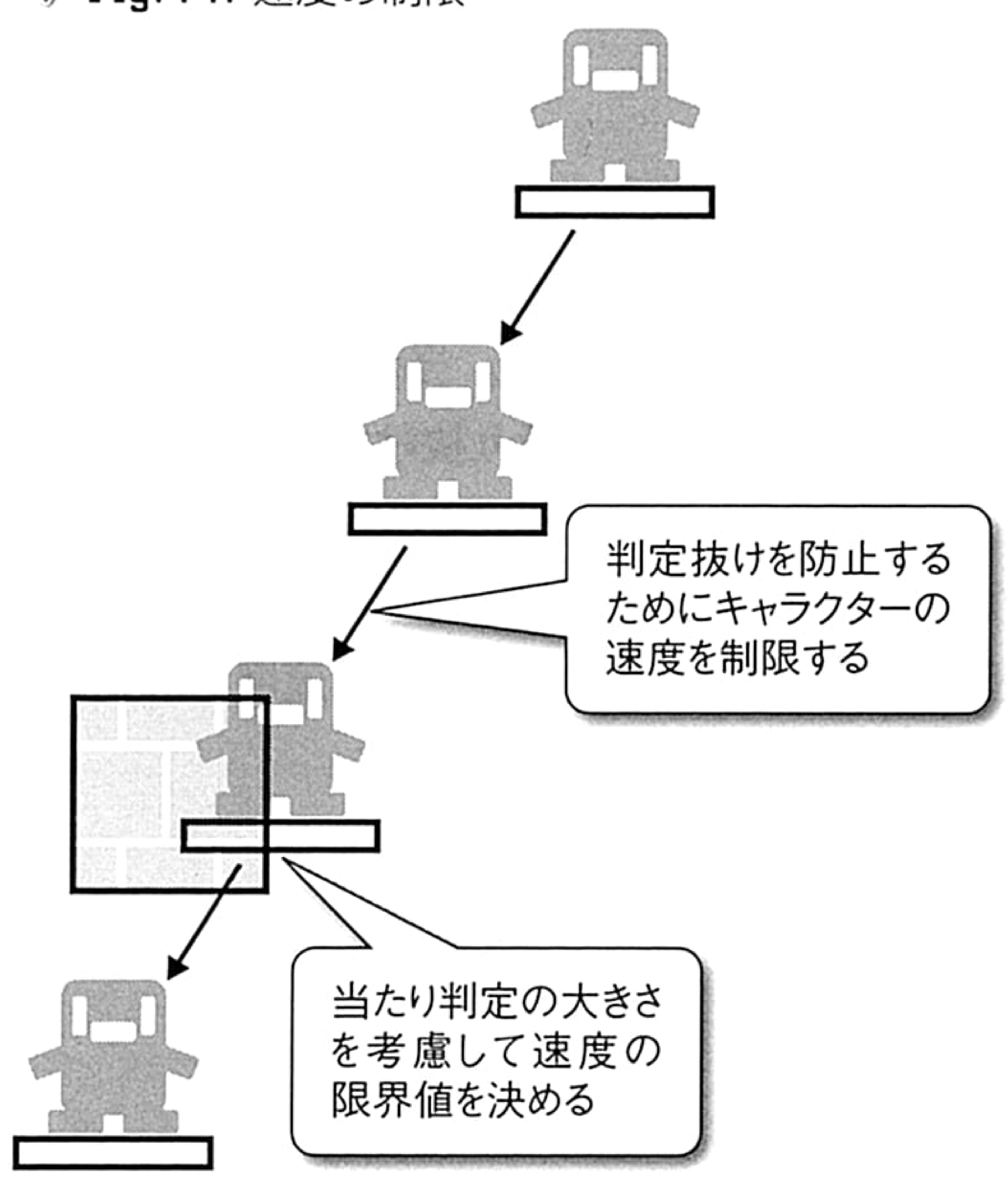
本書のサンプルでは矩形を使った当たり判定処理を使いますが、この方法ではキャラクターの移動速度が大きすぎると、判定抜けが起こることがあります(Fig.43)。速度が大きいために、当たり判定同士が接触することなく、対象物を通り抜けてしまうのです。

▼ **Fig.43**: 判定抜け



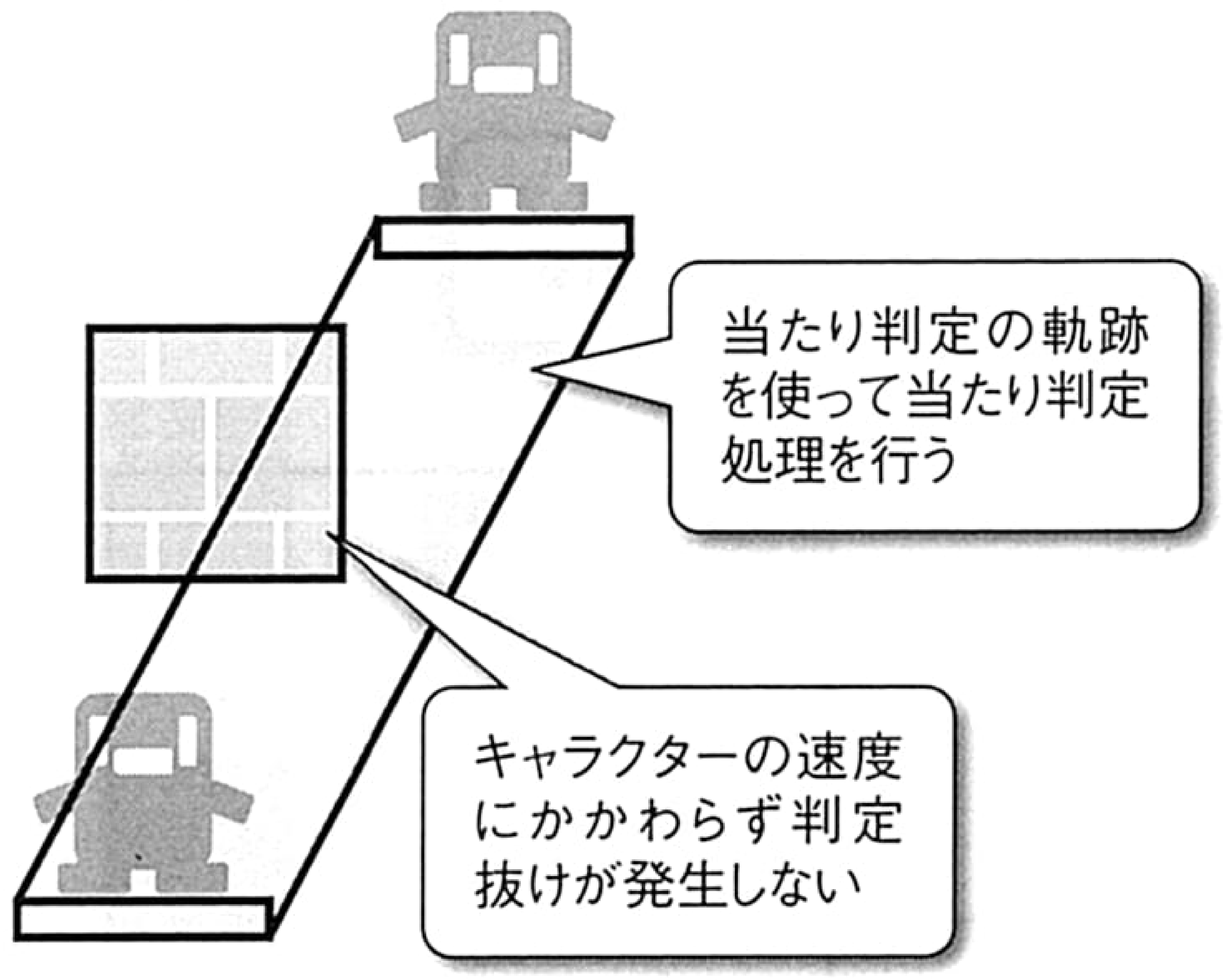
例えば自機が落下するときには、重力に引かれてだんだん速度が大きくなります。判定抜けを防止するには、この速度が大きくなりすぎないように、一定値以下に制限する必要があります(Fig.44)。速度の限界値は、自機と床の当たり判定の大きさを考慮して、判定抜けが起きないように決めます。

▼ **Fig.44**: 速度の制限



判定抜けを防止するもう一つの方法は、キャラクターの当たり判定が通過した軌跡を使って、当たり判定処理を行うことです(Fig.45)。本書では用いませんが、この方法は3Dゲームなどでよく使われています。

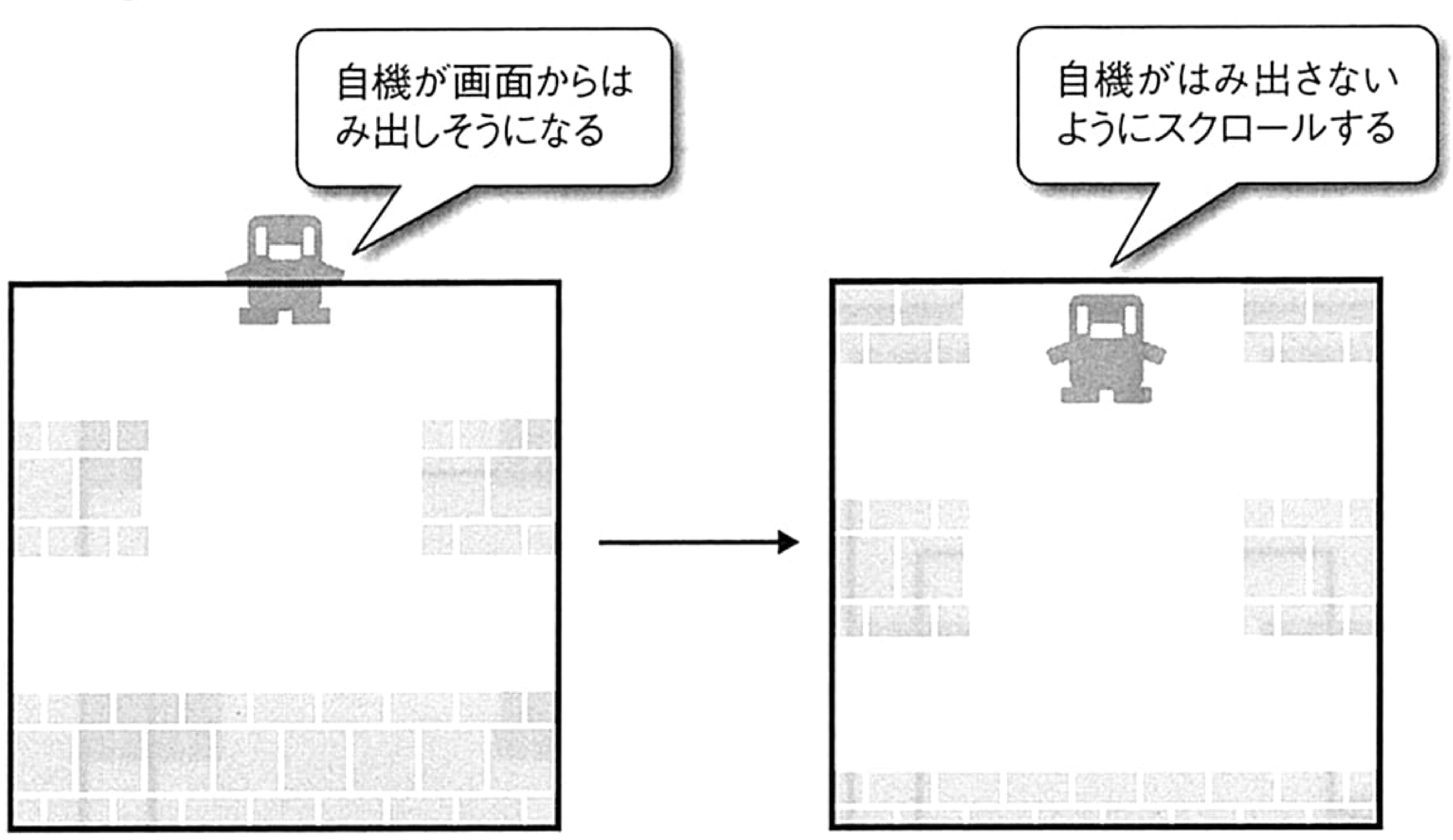
▼ Fig.45: 当たり判定の軌跡を使う



自機の動きに合わせたスクロール

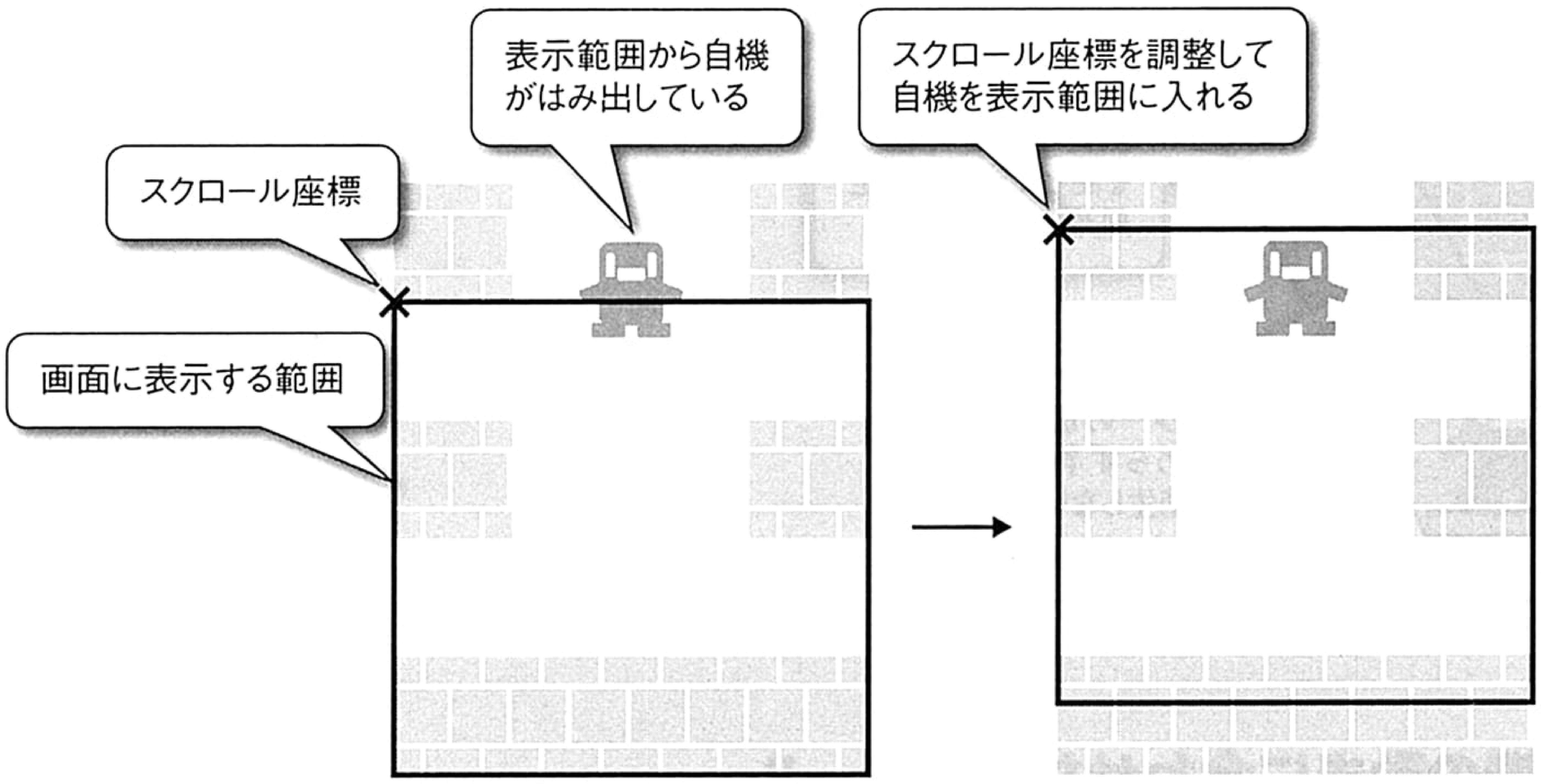
床を押し上げるサンプルでは、自機の動きに合わせてフィールドがスクロールします(Fig.46)。第7章や第8章のサンプルもスクロールを行いますが、自機は常に画面の中央にいました。今回のサンプルでは、自機が画面からはみ出しそうになったときだけスクロールします。

▼ Fig.46: 自機の動きに合わせたスクロール



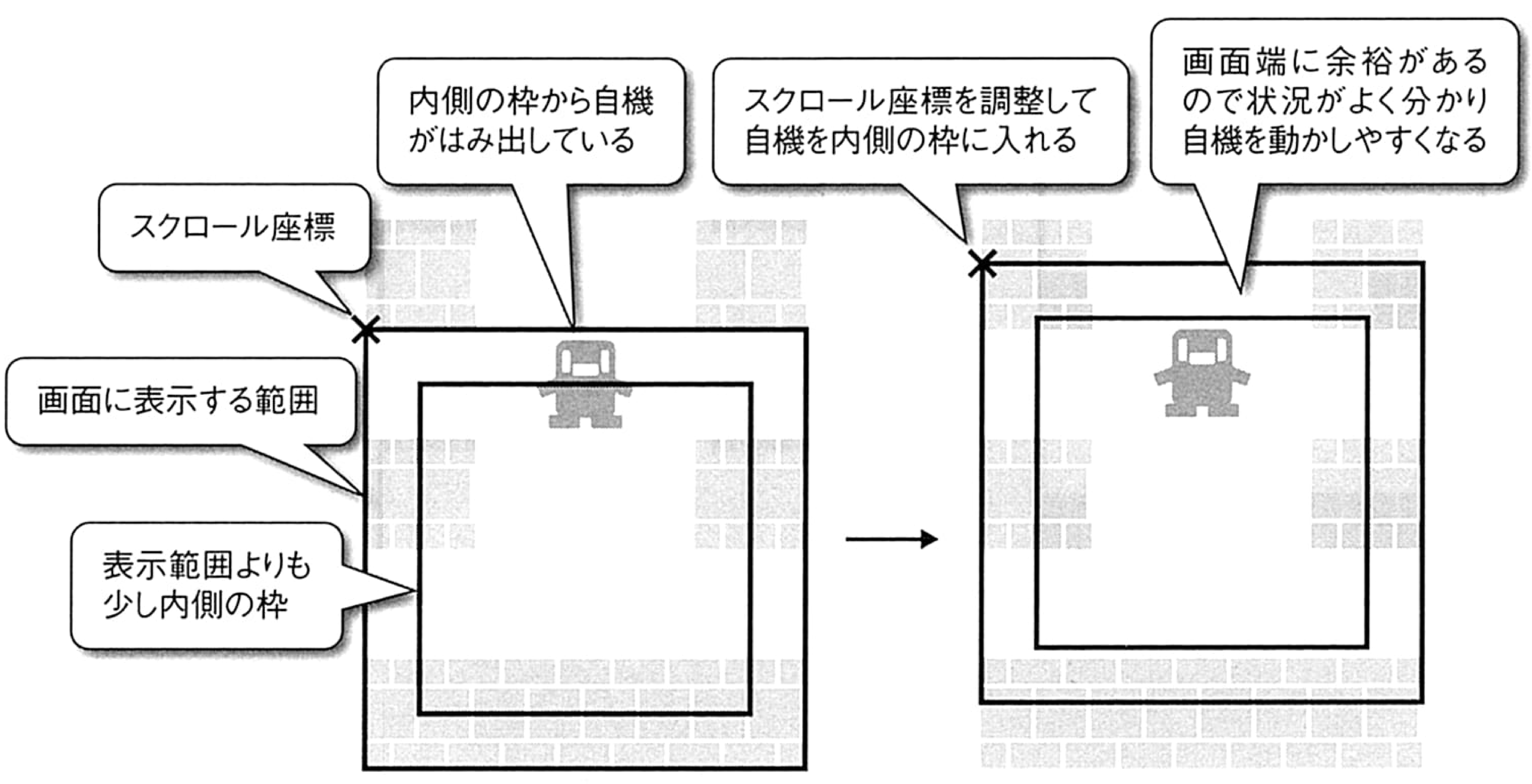
このスクロールを実現するには、フィールドのスクロール座標(p.191)を使って、画面に表示される範囲を調べます(Fig.47)。この表示範囲から自機がはみ出していたら、自機が画面に入るようにスクロール座標を変更します。例えば自機が表示範囲の上側にあったら、スクロール座標を上側に移動して、自機を表示範囲に入れます。

▼ **Fig.47**: 自機を表示範囲に入れる



自機が画面にぎりぎり入るようにスクロールするよりも、自機が画面のやや内側に入るように、余裕を持たせるとよいでしょう(Fig.48)。これはスクロール座標を調整するときに、自機が画面の表示範囲よりも少し内側の枠に入るようにすれば実現できます。余裕があることで画面端の状況がよく分かり、自機を動かしやすくなります。

▼ **Fig.48**: 自機を画面の少し内側に入れる



床を押し上げる自機のプログラム

　List7は自機のプログラムです。自機には着地状態とジャンプ状態があります。着地状態では、

```
if (game.isKey(KEY_SPACE)) {
    obj.vy=-PLAYER_VY;
    obj.jump=true;
}
```

のように、スペースキーを押したかどうかを調べます。スペースキーを押したら、上方向の速度を設定して、ジャンプ状態に移行します。

　ジャンプ状態では、自機が重力に引かれて落下する様子を再現するために、

```
obj.vy+=PLAYER_AY;
```

のように、Y方向の速度に加速度を加えます。また、判定抜けを防止するために、

```
if (obj.vy>PLAYER_VY) {
    obj.vy=PLAYER_VY;
}
```

のように、Y方向の速度を制限します。

　自機と床(または壁)の間では、いくつかの当たり判定処理を行います。まず、

```
obj.sprite.initHit(playerBodyHit);
```

のように全身の当たり判定を使って当たり判定処理を行い、

```
if (game.testHitSprite(obj.sprite)) {
    obj.vx=0;
}
```

のように、床(壁)に接触したらX方向の速度を0にして、左右への移動を中止します。

　次に、自機が上昇中の場合には、

```
obj.sprite.initHit(playerHeadHit);
```

のように頭の当たり判定を使って、床を押し上げたかどうかを判定します。床に接触したら、

```
if (spr.getState()==STATE_WALL_STAY) {
    spr.setState(STATE_WALL_LIFT);
}
```

のように床を押し上げ状態にした上で、

```
obj.vy=0;
```

のようにY方向の速度を0にして、自機を落下させます。
　続いて、自機が下降中の場合には、

```
obj.jump=true;
```

として、仮にジャンプ状態にしておいてから、

```
obj.sprite.initHit(playerFootHit);
```

のように足の当たり判定を使って、床に着地したかどうかを判定します。仮にジャンプ状態にしておくのは、床から外れたときに落下させるためです。床に接触したら、

```
obj.vy=(spr.getY()-1)-(obj.sprite.getY()-obj.vy);
```

として、床の上に自機がぴったり乗るように、Y方向の速度を設定することによって、自機のY座標を調整します。Y座標を設定せずに速度を使って間接的に設定するのは、プログラムの流れの都合なので、直接Y座標を設定する処理にしても構いません。さらに、

```
obj.jump=false;
```

として、自機を着地状態にします。
　また、自機の動きに合わせてフィールドをスクロールするために、

```
gx=game.getViewX();
gy=game.getViewY();
```

のようにスクロール座標を取得します。そして、

```
if (x<gx+1) gx=x-1; else if (x>gx+XMAX-1) gx=x-XMAX+1;
if (y<gy+1) gy=y-1; else if (y>gy+YMAX-1) gy=y-YMAX+1;
```

のように、自機が表示範囲の左右または上下から出ていたら、自機が表示範囲に入るようにスクロール座標を調整します。最後に、

```
game.setViewXY(gx, gy);
```

として、調整したスクロール座標を設定します。

List7: 床を押し上げる自機(liftFloor.js)

```
// 自機のX方向の速度の大きさ
var PLAYER_VX=0.2;

// 自機のY方向の速度の大きさ
var PLAYER_VY=0.6;

// 自機のY方向の加速度の大きさ
var PLAYER_AY=0.05;

// 自機の当たり判定(全身・頭・足)
var playerBodyHit;
var playerHeadHit;
var playerFootHit;

// 自機
function Player(x, y) {
  var obj=this;

  // スプライトの作成
  obj.sprite=game.createSprite();

  // アニメーション・座標・速度の設定
  obj.sprite.initAnim(playerWalkAnim);
  obj.sprite.setXY(x, y);
  obj.vx=0;
  obj.vy=0;

  // ジャンプ状態(true)か着地状態(false)か
  obj.jump=false;

  // 更新処理
```

```
obj.sprite.onUpdate=function() {

  // 座標と速度の取得
  var x=obj.sprite.getX(), y=obj.sprite.getY();
  var gx, gy;

  // カーソルキーの入力に応じて左右に移動する
  obj.vx=0;
  if (game.isKey(KEY_LEFT)) obj.vx=-PLAYER_VX;
  if (game.isKey(KEY_RIGHT)) obj.vx=PLAYER_VX;

  // 着地状態
  if (!obj.jump) {

    // Y方向の速度を0にする
    obj.vy=0;

    // スペースキーを押したら、
    // 上方向の速度を設定し、ジャンプ状態にする
    if (game.isKey(KEY_SPACE)) {
      obj.vy=-PLAYER_VY;
      obj.jump=true;
    }
  } else

  // ジャンプ状態
  {
    // Y方向の速度に加速度を加える
    obj.vy+=PLAYER_AY;

    // 当たり判定処理で抜けが生じないように、
    // Y方向の速度を制限する
    if (obj.vy>PLAYER_VY) {
      obj.vy=PLAYER_VY;
    }
  }

  // 全身の当たり判定を使って、
  // 左右の壁に当たったかどうかを判定する
  obj.sprite.setXY(x+obj.vx, y+obj.vy);
  obj.sprite.initHit(playerBodyHit);
  obj.sprite.onHitSprite=function() {}
  if (game.testHitSprite(obj.sprite)) {

    // 接触したら、X方向の速度を0にする
    obj.vx=0;
  }

  // 上昇中の処理
```

```
if (obj.vy<0) {

  // 頭の当たり判定を使って、
  // 床を押し上げたかどうかを判定する
  obj.sprite.setXY(x+obj.vx, y+obj.vy);
  obj.sprite.initHit(playerHeadHit);
  obj.sprite.onHitSprite=function(spr) {

    // 静止状態の床に接触したら、
    // 床を押し上げ状態にする
    if (spr.getState()==STATE_WALL_STAY) {
      spr.setState(STATE_WALL_LIFT);
    }

    // Y方向の速度を0にすることによって、
    // 自機を落下させる
    obj.vy=0;
  }
  game.testHitSprite(obj.sprite);
} else

// 下降中の処理
{
  // 足の当たり判定を使って、
  // 床に着地しているかどうかを判定する
  obj.sprite.setXY(x+obj.vx, y+obj.vy);
  obj.jump=true;
  obj.sprite.initHit(playerFootHit);
  obj.sprite.onHitSprite=function(spr) {

    // ジャンプ状態で床に接触したら、着地する
    if (obj.jump) {

      // 床の上に自機がちょうどよく乗るように、
      // Y方向の速度を調整することにより、
      // Y座標を調整する
      obj.vy=(spr.getY()-1)-(obj.sprite.getY()-obj.vy);

      // 着地状態にする
      obj.jump=false;
    }
  }
  game.testHitSprite(obj.sprite);
}

// 座標の更新
x+=obj.vx;
y+=obj.vy;
obj.sprite.setXY(x, y);
```

```
      // 着地状態で左右に移動しているときには、
      // アニメーションを再生する
      if (!obj.jump && obj.vx!=0) {
        obj.sprite.stepAnim();
      }

      // スクロール座標の取得
      gx=game.getViewX();
      gy=game.getViewY();

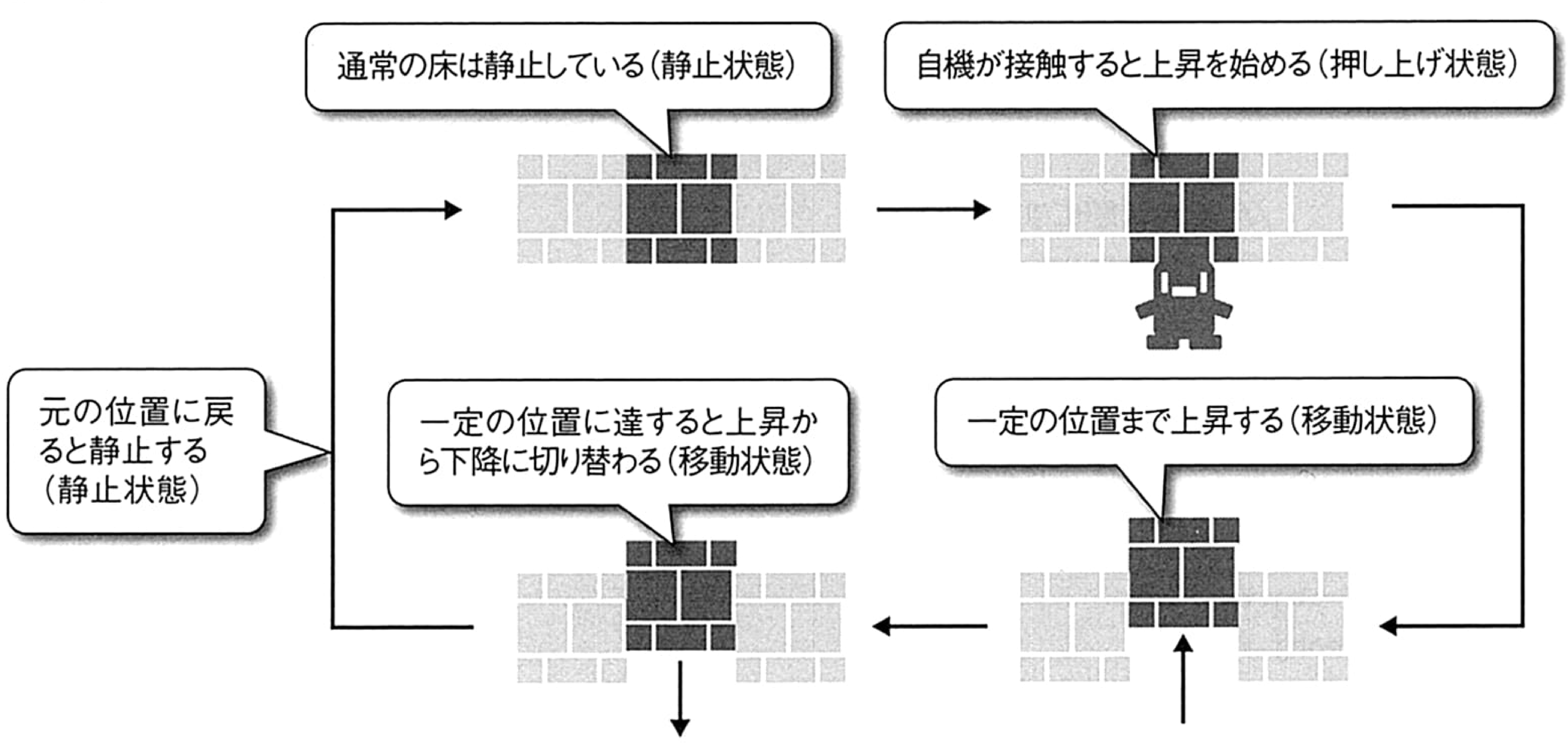
      // 自機が表示範囲から出ていたら、
      // 自機が表示範囲に入るようにスクロール座標を調整する
      if (x<gx+1) gx=x-1; else if (x>gx+XMAX-1) gx=x-XMAX+1;
      if (y<gy+1) gy=y-1; else if (y>gy+YMAX-1) gy=y-YMAX+1;

      // スクロール座標の更新
      game.setViewXY(gx, gy);
    }
  }
```

押し上がる床

自機が押し上げる床は、普段は静止していますが、自機が押し上げると上昇し、一定の位置まで上昇すると下降に切り替わって、元の位置に戻ると静止します。この動きを実現するために、静止状態・押し上げ状態・移動状態という3つの状態を設けます(Fig.49)。

Fig.49: 床の状態



静止状態は止まっている状態です。静止状態の床を自機を押し上げると、押し上げ状態になります。押し上げ状態では上方向の速度を設定して、移動状態に移行します。移動状態では、下方向の加速度を加えることで、床をだんだん落下させます。最初の位置まで落下したら、静止状態に戻ります。

静止状態と移動状態の間に押し上げ状態があるのは、プログラムの都合です。このプログラムでは、自機オブジェクトが床を押し上げ状態に設定してから、床オブジェクトが上方向の速度などを設定する方法をとっています。自機オブジェクトが床の速度などを直接設定すれば、押し上げ状態は省略することができます。どちらも結果は同じなので、プログラムが分かりやすくなる方式を使うとよいでしょう。

押し上がる床のプログラム

List8は床のプログラムです。床の処理はWallオブジェクトにまとめました。なお、床はwallよりもむしろfloorですが、壁と共通の画像(wall.png)を使っているので、ここではWallオブジェクトにしました。

最初は、

```
obj.sprite.setState(STATE_WALL_STAY);
```

のように、床を静止状態にしておきます。自機が床を押し上げると、押し上げ状態になります。

押し上げ状態では、

```
vy=-WALL_VY;
```

のように上方向の速度を設定し、

```
obj.sprite.setState(STATE_WALL_MOVE);
```

として移動状態に移行します。

移動状態では、

```
y+=vy;
vy+=WALL_AY;
```

のように座標と速度を更新します。速度に加速度を加えることによって、床は上昇から下降に転じます。床が最初の位置まで落下したら、

```
if (y>=obj.initY) {
    y=obj.initY;
    obj.sprite.setState(STATE_WALL_STAY);
}
```

のように、床のY座標を初期値に戻し、静止状態に移行します。

▼ **List8**: 押し上がる床(liftFloor.js)

```
// 床のY方向の速度の大きさ
var WALL_VY=0.2;

// 床のY方向の加速度の大きさ
var WALL_AY=0.05;

// 床の状態(静止・押し上げ・移動)
var STATE_WALL_STAY=1;
var STATE_WALL_LIFT=2;
var STATE_WALL_MOVE=3;

// 床
function Wall(x, y) {
  var obj=this;

  // スプライトの作成
  obj.sprite=game.createSprite();

  // アニメーションと座標の設定
  obj.sprite.initAnim(wallAnim);
  obj.sprite.setXY(x, y);

  // 最初は静止状態にする
  obj.sprite.setState(STATE_WALL_STAY);

  // 初期座標と速度の設定
  obj.initX=x;
  obj.initY=y;
  obj.vy=0;

  // 更新処理
  obj.sprite.onUpdate=function() {

    // 座標と速度の取得
```

```
    var x=obj.sprite.getX(), y=obj.sprite.getY();
    var vy=obj.vy;

    // 押し上げ状態
    if (obj.sprite.getState()==STATE_WALL_LIFT) {

      // 上方向の速度を設定する
      vy=-WALL_VY;

      // 移動状態に移行する
      obj.sprite.setState(STATE_WALL_MOVE);
    }

    // 移動状態
    if (obj.sprite.getState()==STATE_WALL_MOVE) {

      // 座標の更新
      y+=vy;

      // 速度に加速度を加える
      vy+=WALL_AY;

      // 初期座標まで戻ってきたら、
      // 静止状態に移行する
      if (y>=obj.initY) {

        // 座標を初期座標にする
        y=obj.initY;

        // 静止状態に移行する
        obj.sprite.setState(STATE_WALL_STAY);
      }
    }

    // 座標と速度の更新
    obj.sprite.setXY(x, y);
    obj.vy=vy;
  }
}
```

床を押し上げるゲームの本体

List9は床を押し上げるゲームの本体です。おなじみの初期化処理(init関数)とメインルーチン(main関数)を定義します。

メインルーチンでは、

```
playerBodyHit=new Array(0.1, 0.1, 0.9, 0.9);
playerHeadHit=new Array(0.1, 0.3, 0.9, 0.4);
playerFootHit=new Array(0.1, 1, 0.9, 1.1);
```

のように、自機の全身·頭·足の当たり判定を作成します。配列の要素は、順に当たり判定の左·上·右·下の座標を表します。全身の当たり判定は自機全体に、頭の当たり判定は自機の上半身に、足の当たり判定は自機の下端にあります(p.293)。

▼ **List9**: 床を押し上げるゲームの本体(liftFloor.js)

```
// セルの種類(空間·壁·自機)
var CELL_VOID=0;
var CELL_WALL=1;
var CELL_PLAYER=2;

// ゲーム本体·自機·壁·フィールドのオブジェクト
var game;
var player;
var wall;
var field;

// 自機と壁のアニメーション
var playerWalkAnim;
var wallAnim;

// 初期化
function init() {
  var x, y, i;

  // 画面の消去
  game.clear();

  // フィールドの作成
  field=game.createField();

  // セルの種類に対応する画像の一覧を配列で設定する
```

```
  field.initCellImage(new Array(
    "void.png", "void.png", "void.png"
  ));

  // セルの初期化
  field.initCell(new Array(
    "111111111111",
    "100000000001",
    "100000000001",
    "100001100001",
    "100000000001",
    "111000000111",
    "100000000001",
    "100001100001",
    "100000000001",
    "111000000111",
    "100000000001",
    "100001100001",
    "100000000001",
    "100002000001",
    "111111111111"
  ));

  // 自機と床を生成する
  wall=new Array();
  for (i=0, y=0; y<field.getYCount(); y++) {
    for (x=0; x<field.getXCount(); x++) {
      switch (field.getCell(x, y)) {

        // 自機の生成
        case CELL_PLAYER:
          player=new Player(x, y);
          break;

        // 床の生成
        case CELL_WALL:
          wall[i++]=new Wall(x, y);
          break;
      }
    }
  }
}

// メインルーチン
function main() {

  // ゲーム本体の作成
```

```
  game=new Game();

  // アニメーションの作成
  playerWalkAnim=new Array(
    "pcWalk0.png", "pcWalk2.png", "pcWalk0.png",
    "pcWalk1.png", "pcWalk3.png", "pcWalk1.png"
  );
  wallAnim=new Array("wall.png");

  // 当たり判定の作成
  playerBodyHit=new Array(0.1, 0.1, 0.9, 0.9);
  playerHeadHit=new Array(0.1, 0.3, 0.9, 0.4);
  playerFootHit=new Array(0.1, 1, 0.9, 1.1);

  // ゲームの初期化
  init();
}
```

Chapter 10

画面の遷移

ゲームの色々な画面

ここまでのサンプルでは、起動するといきなりゲームが始まりました。また、障害物や敵に当たってゲームオーバーになると、そのまま新しいゲームに移りました。

多くのゲームにはタイトル画面やゲームオーバー画面があります。一般に、起動するとタイトル画面が表示され、キーやマウスボタンを入力するとゲームが始まります。ゲームが終わるとゲームオーバー画面が表示されて、一定時間が経過するとタイトル画面に戻ります。

タイトル画面やゲームオーバー画面以外にも、ゲームには次のように色々な画面があります。

- **タイトル画面**
  ゲームの起動時に、ゲームの題名を表示する画面です。
- **レディ画面**
  ステージが始まる前に、プレイヤーに準備を促す画面です。
- **クリア画面**
  ステージを完了したことを示す画面です。
- **コンティニュー画面**
  ゲームが終わったときに、ゲームを継続するか、終了するかを尋ねる画面です。
- **ゲームオーバー画面**
  ゲームが完全に終了したときに表示される画面です。

こういった画面があると、作品としての体裁が整うだけでなく、ゲームが遊びやすくなります。そこで本章では、第8章のアイテムを拾うゲーム(p.224)に対して色々な画面を追加しました。ゲームを実行するにはgameUI.htmlをブラウザで開きます。

ゲームを起動するとタイトル画面が表示されます(Fig.1)。「Press [Enter] Key To Start」というメッセージは、時間とともに点滅します。

▼ Fig.1：タイトル画面

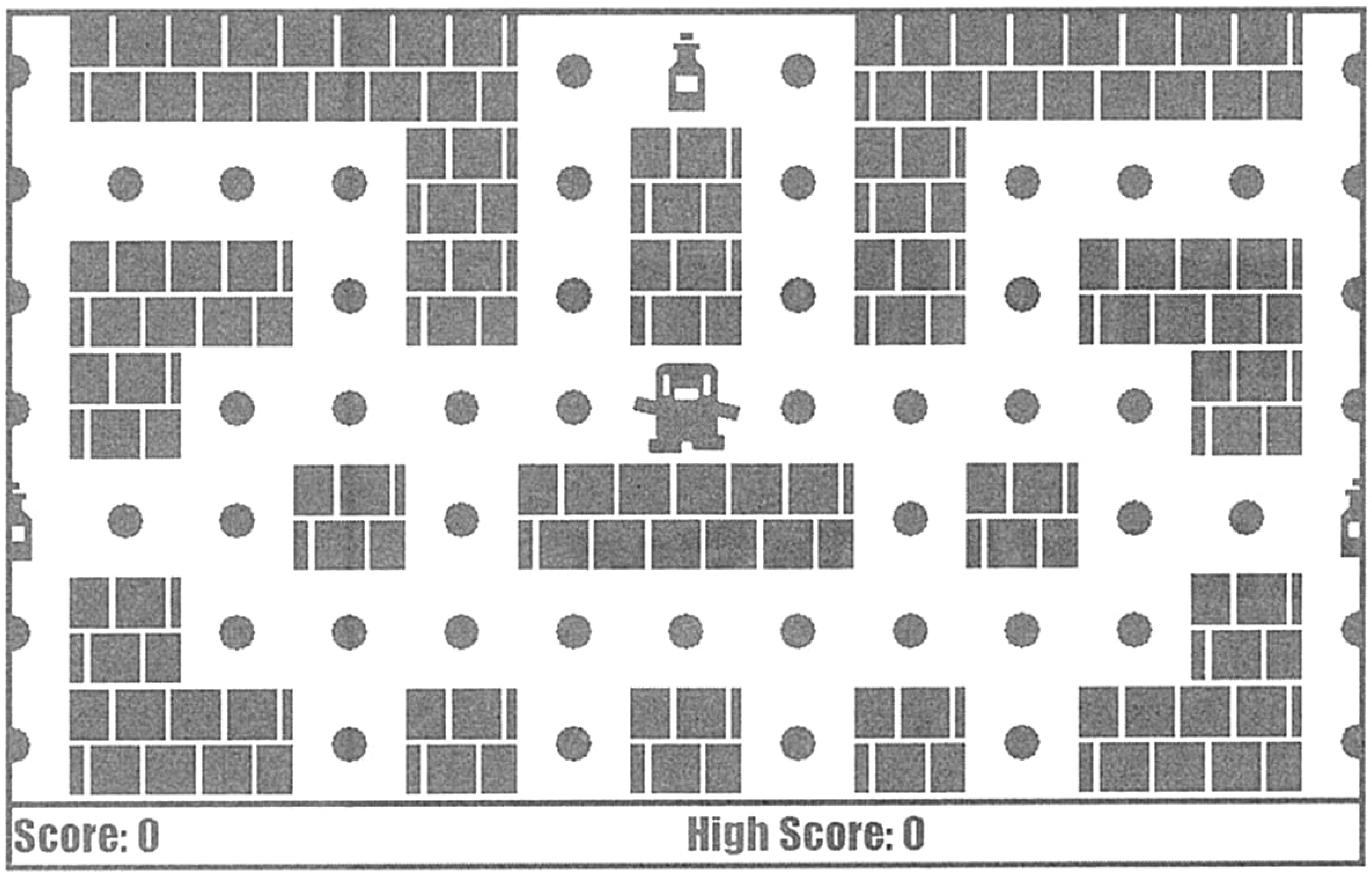


タイトル画面でEnterキーを押すと、レディ画面になります(Fig.2)。一定時間が経過すると、ゲームが始まります(Fig.3)。

▼ Fig.2：レディ画面

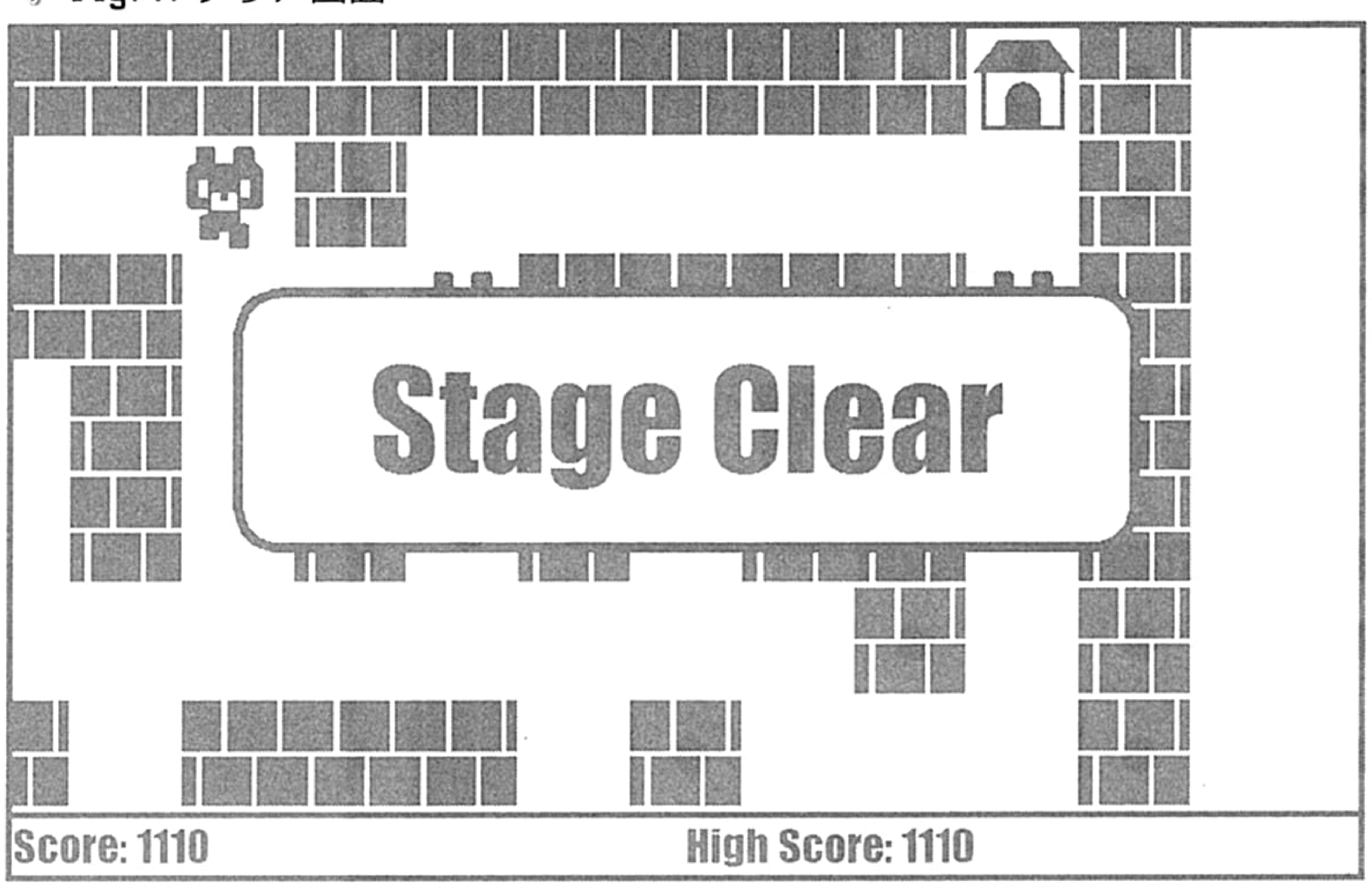


▼ Fig.3: ゲームの開始



ステージをクリアすると、クリア画面になります(Fig.4)。一定時間が経過すると、レディ画面が表示された後に、次のステージが始まります。

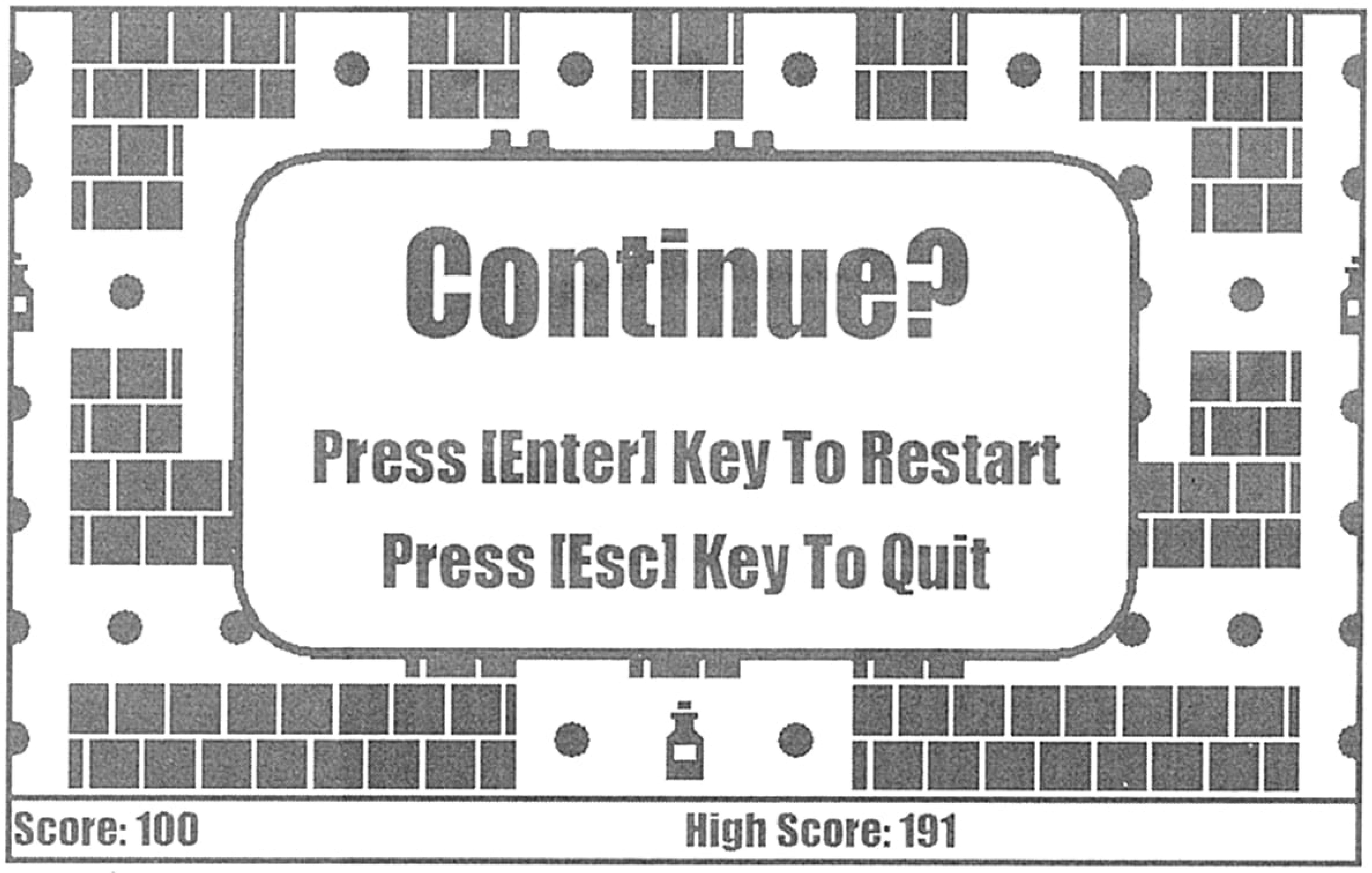
▼ Fig.4: クリア画面



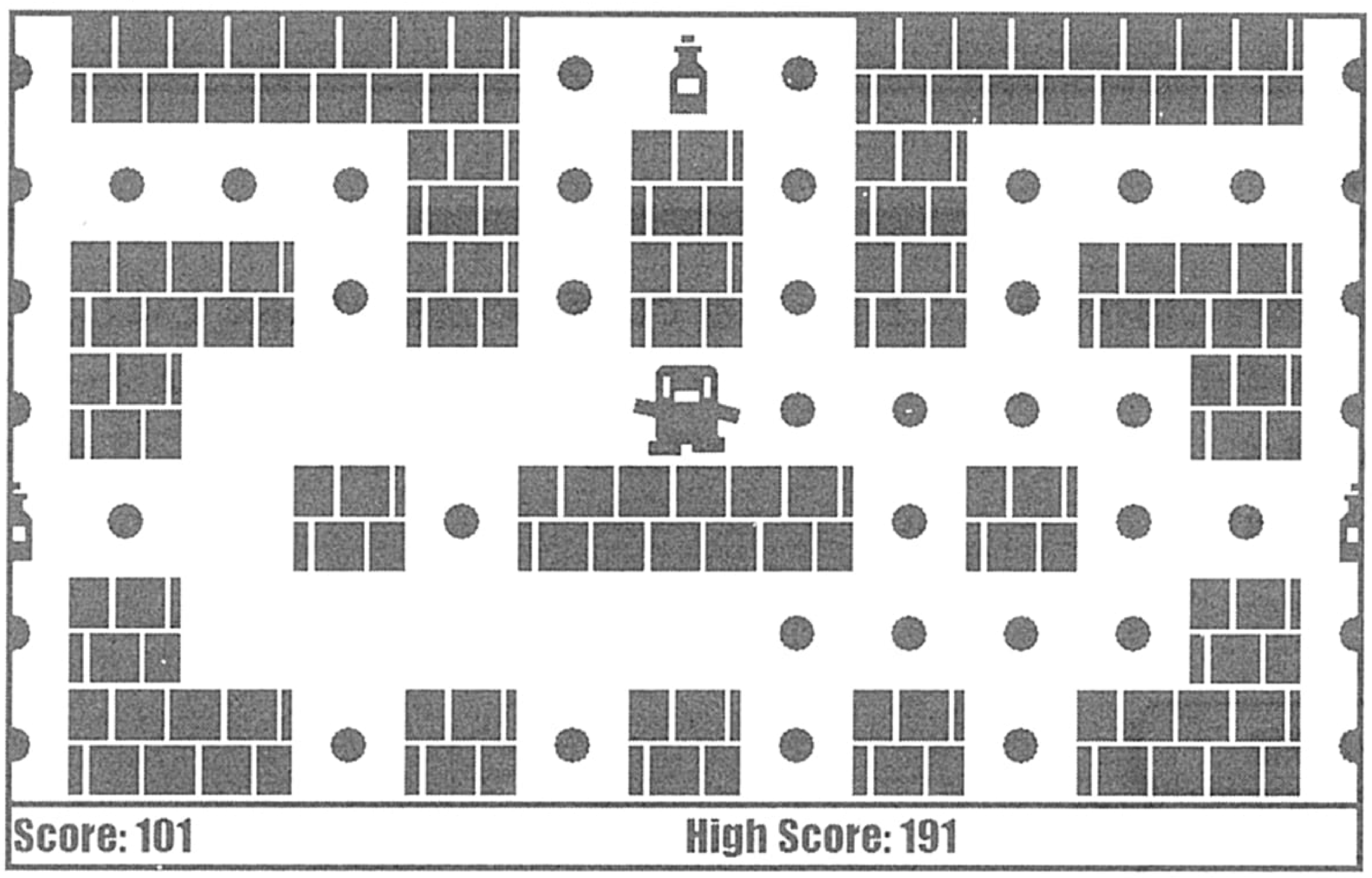
自機が敵に接触するとゲーム終了となり、コンティニュー画面が表示されます(Fig.5)。Enterキーを押すと、ゲームを再開することができます(Fig.6)。

なお、コンティニュー(ゲームの再開)をするとスコアに1点が加算されます。スコアの下1桁を見れば、何回コンティニューをしたのかが分かります。

▼ Fig.5: コンティニュー画面

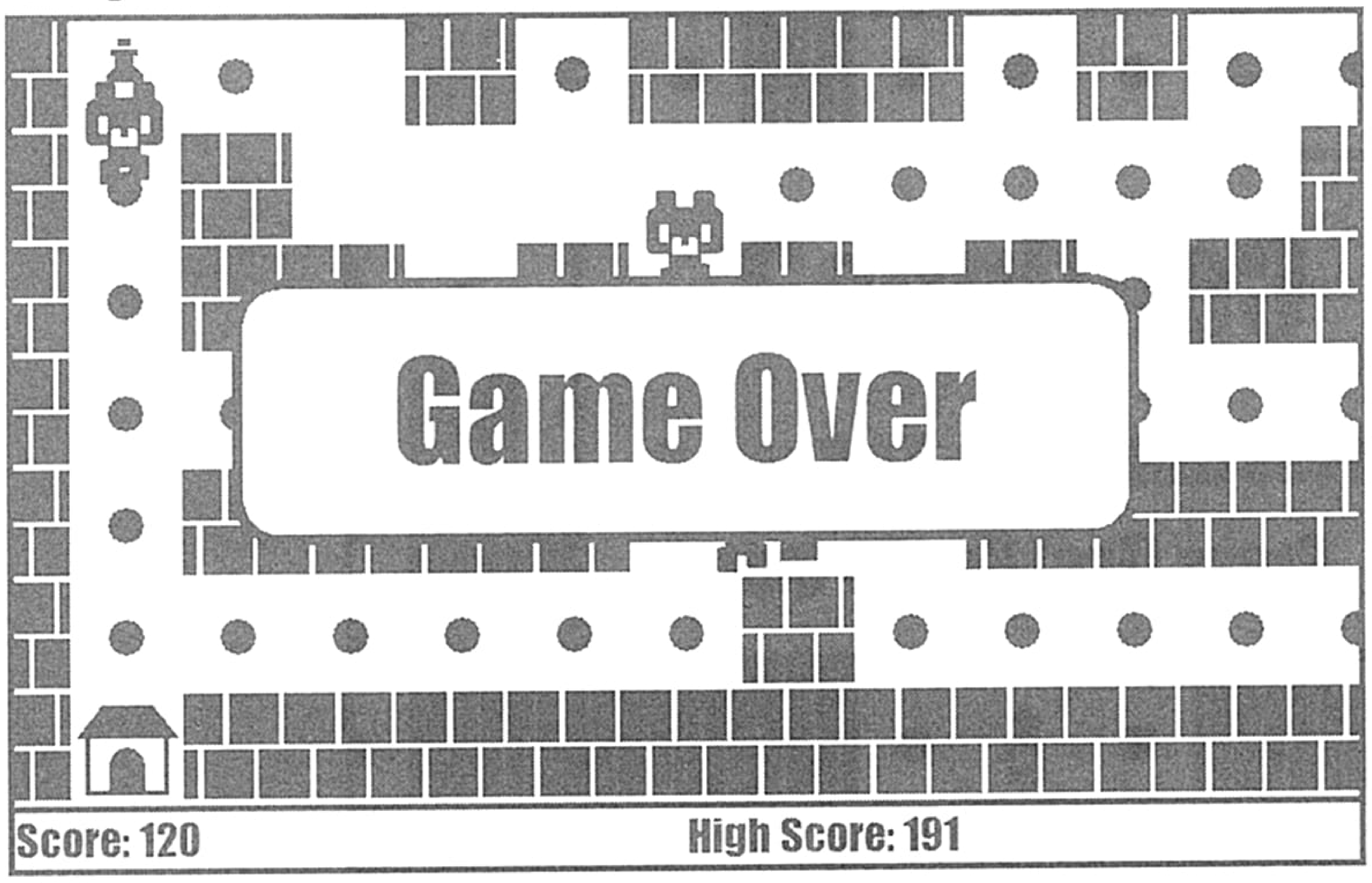


▼ Fig.6: ゲームの再開



コンティニュー画面でEscキーを押すと、ゲームオーバー画面が表示されます(Fig.6)。一定時間が経過すると、タイトル画面に戻ります。

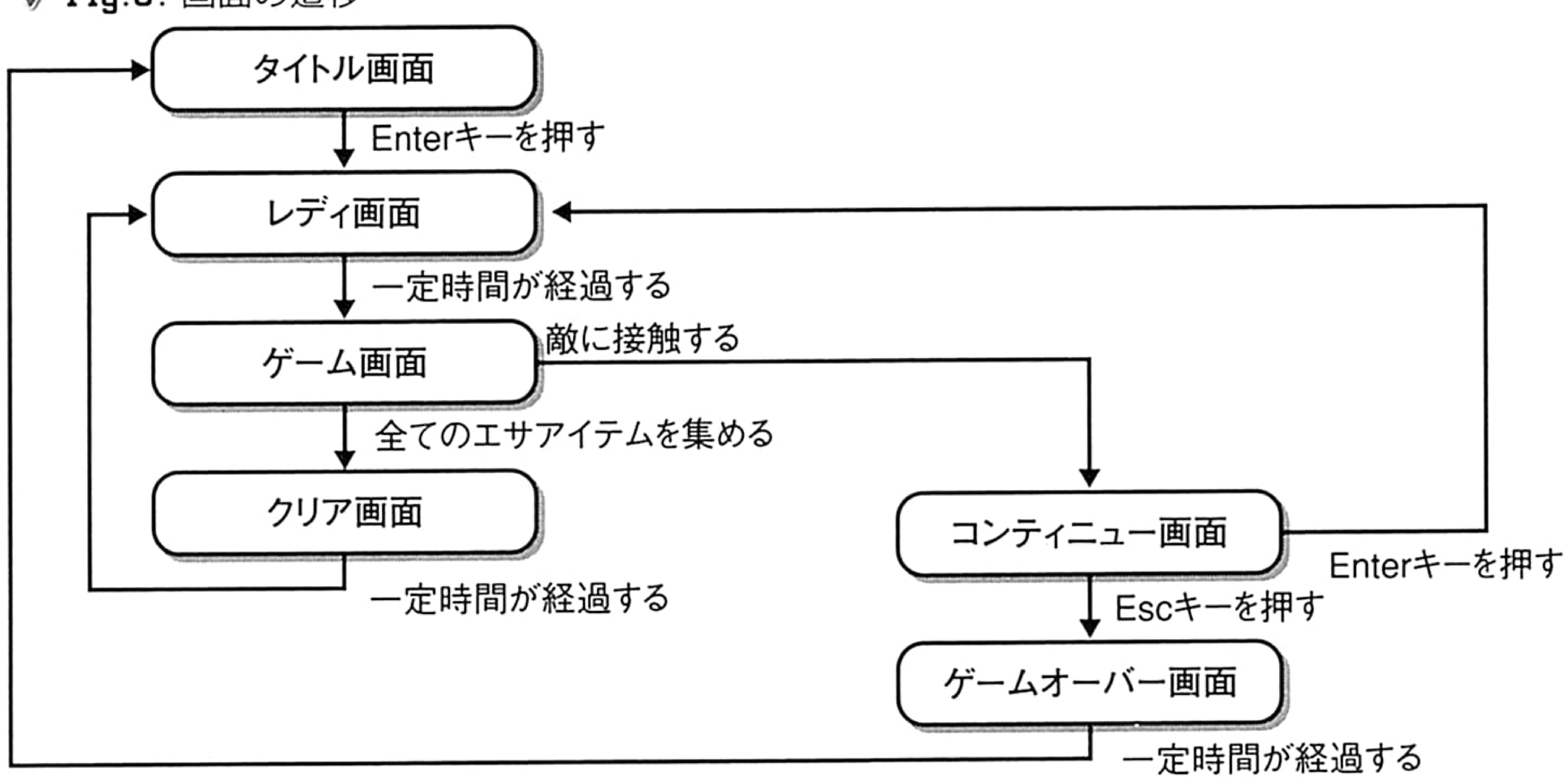
▼ Fig.7: ゲームオーバー画面



画面の遷移

ある画面から別の画面に切り替えることを、本書では画面の遷移と呼ぶことにしましょう。サンプルにはタイトル画面・レディ画面・クリア画面・コンティニュー画面・ゲームオーバー画面といった画面があります。キーの入力や一定時間の経過といった条件によって、これらの画面の間を遷移します(Fig.8)。

▼ Fig.8: 画面の遷移



List1は画面の遷移を実現するプログラムです。自機や敵の処理は第8章と共通なので、ここでは画面の遷移に関する部分を中心に紹介します。

まず、画面の状態を表す変数game.stateを用意します。この変数には、現在表示している画面の種類を表す定数を格納します。例えばタイトル画面の定数はGS_TITLE、レディ画面の定数はGS_READYです。

メインルーチン(main関数)では、

```
game.state=GS_TITLE;
```

として、最初はタイトル画面が表示されるようにしておきます。また、

```
game.onUpdate=function() {
```

のように、GameオブジェクトのonUpdate関数(p.67)を定義します。この関数はSpriteオブジェクトのonUpdate関数と同様に、一定時間(50ミリ秒)ごとに呼び出されます。

onUpdate関数を定義することによって、ゲームごとに異なる独自の処理を行うことができます。ここでは、

```
switch (game.state) {
```

のようなswitch文を記述します。変数game.stateの値を使って分岐し、現在表示している画面の種類に応じた処理を行います。

例えばタイトル画面の場合には、

```
case GS_TITLE:
    initTitle();
    game.state=GS_TITLE2;
```

として、タイトル画面の初期化と、新しい状態(GS_TITLE2)への遷移を行います。各画面ごとの詳細な処理については後述します。

自機のオブジェクト(Playerオブジェクト)と敵のオブジェクト(Enemyオブジェクト)については、各オブジェクトのonUpdate関数に、

```
if (game.state!=GS_GAME) return;
```

のようなif文を追加しました。GS_GAMEはゲーム画面(ゲーム中の画面)を表す定数です。

このif文によって、自機や敵はゲーム中のみ動き、レディ画面やコンティニュー画面などでは動かなくなります。これらの画面では自機や敵を止めておきたいので、このような処理を行います。

List1: 画面の遷移(gameUI.js)

```
// 現在の画面を表す定数
//(タイトル·レディ·ゲーム·クリア·コンティニュー·ゲームオーバー)
var GS_TITLE=0;
var GS_TITLE2=1;
var GS_READY=2;
var GS_READY2=3;
var GS_GAME=4;
var GS_CLEAR=5;
var GS_CLEAR2=6;
var GS_CONTINUE=7;
var GS_CONTINUE2=8;
var GS_GAMEOVER=9;
var GS_GAMEOVER2=10;

// 自機
function Player(x, y) {
  // ...(中略)...

  // 更新処理
  obj.sprite.onUpdate=function() {

    // ゲーム画面の場合だけ自機を動かす
    if (game.state!=GS_GAME) return;

    // ...(中略)...
  }
}

// 敵
function Enemy(x, y) {
  // ...(中略)...

  // 更新処理
  obj.sprite.onUpdate=function() {

    // ゲーム画面の場合だけ敵を動かす
    if (game.state!=GS_GAME) return;

    // ...(中略)...
  }
}

// メインルーチン
```

```
function main() {
  // ...(中略)...

  // 更新処理
  game.onUpdate=function() {

    // 画面に応じて分岐する
    switch (game.state) {

      // タイトル画面の初期化
      case GS_TITLE:
        initTitle();
        game.state=GS_TITLE2;

      // タイトル画面の本体
      case GS_TITLE2:
        title.stepAnim();
        if (game.isKey(KEY_ENTER)) {
          initGame();
          game.state=GS_READY;
        }
        break;

      // レディ画面の初期化
      case GS_READY:
        initSplash("ready.png");
        game.time=0;
        game.state=GS_READY2;

      // レディ画面の本体
      case GS_READY2:
        game.time++;
        if (game.time==20) {
          initSplash("void.png");
          game.state=GS_GAME;
        }
        break;

      // ...(中略)...
    }
  }

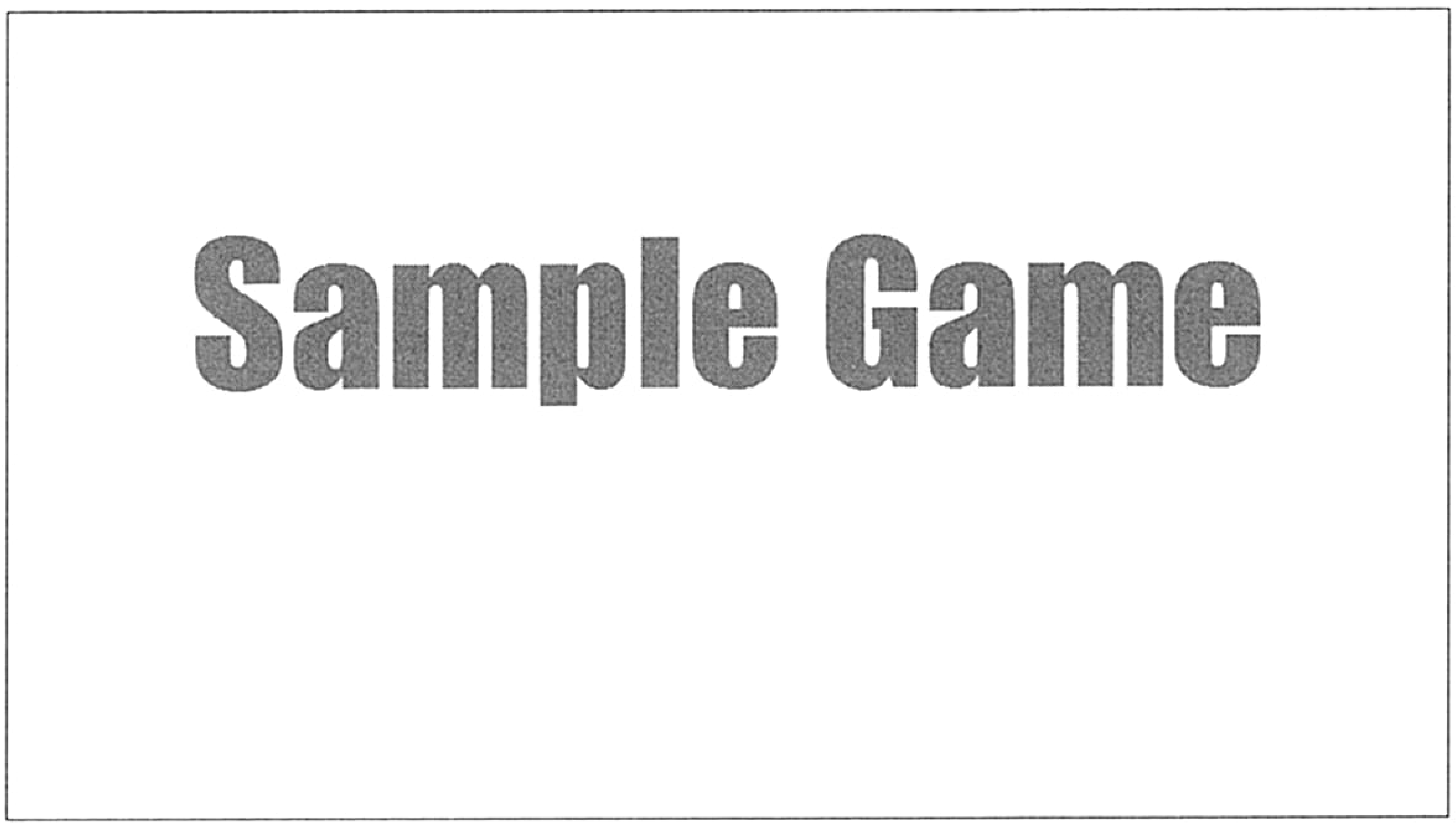
  // 最初はタイトル画面にする
  game.state=GS_TITLE;

  // タイマーの初期化
  game.time=0;
}
```

タイトル画面

個別の画面について、まずはタイトル画面から詳しく見てみましょう。サンプルではタイトル画面のために、3枚の画像を用意しました(Fig.9～11)。

▼ **Fig.9**: タイトル画面(title0.png)



▼ **Fig.10**: タイトル画面(title1.png)



▼ **Fig.11**: タイトル画面(title2.png)



各画像の違いは「Press [Enter] Key To Start」というメッセージです。1枚目にはメッセージがなく、2枚目には灰色でメッセージが入り、3枚目には黒色でメッセージが入っています。これらの画像を組み合わせて、メッセージが現れたり消えたりするようなアニメーションを構成しています。同じ要領で、もっと複雑なアニメーションを行うこともできます。

タイトル画面のプログラム

List2はタイトル画面のプログラムです。タイトル画面を初期化するinitTitle関数と、メインルーチンのmain関数があります。

initTitle関数では、

```
title=game.createSprite();
```

のようにタイトル画面のスプライトを初期化し、

```
title.initAnim(titleAnim);
```

のようにアニメーションを設定します。アニメーションの内容はmain関数で定義します。また、

```
title.setSize(CELL_XCOUNT, CELL_YCOUNT);
```

のようにサイズを画面一杯にします。

main関数で定義する更新処理(onUpdate関数)では、switch文を使って画面の種類ごとに異なる処理を行います。タイトル画面の処理は、

- タイトル画面の初期化(GS_TITLE)
- タイトル画面の本体(GS_TITLE2)

という2つに分けました。

初期化については、

```
initTitle();
```

のように、前述のinitTitle関数を使ってタイトル画面を初期化し、

```
game.state=GS_TITLE2;
```

として、タイトル画面の本体に遷移します。

本体については、

```
title.stepAnim();
```

のようにタイトル画面のアニメーションを再生します。次に、

```
if (game.isKey(KEY_ENTER)) {
    initGame();
    game.state=GS_READY;
}
```

のように、Enterキーの入力を調べて、入力があったらゲームを初期化し、レディ画面(GS_READY)に遷移します。

List2: タイトル画面(gameUI.js)

```
// タイトル画面のスプライト
var title;

// タイトル画面のアニメーション
var titleAnim;

// タイトル画面の初期化
function initTitle() {
```

```
  // 画面の消去
  game.clear();

  // スクロール座標の初期化
  game.setViewXY(0, 0);

  // スプライトの作成
  title=game.createSprite();

  // アニメーションとサイズの設定
  title.initAnim(titleAnim);
  title.setSize(CELL_XCOUNT, CELL_YCOUNT);
}

// メインルーチン
function main() {
  // ...(中略)...

  // タイトル画面のアニメーション
  titleAnim=new Array(
    "title0.png", "title0.png", "title0.png", "title0.png",
    "title1.png", "title1.png",
    "title2.png", "title2.png", "title2.png", "title2.png",
    "title1.png", "title1.png"
  );

  // 更新処理
  game.onUpdate=function() {

    // 画面に応じて分岐する
    switch (game.state) {

      // タイトル画面の初期化
      case GS_TITLE:

        // タイトル画面を作成する
        initTitle();

        // タイトル画面の本体に遷移する
        game.state=GS_TITLE2;

      // タイトル画面の本体
      case GS_TITLE2:

        // タイトル画面のアニメーションを再生する
        title.stepAnim();

        // Enterキーを押したらゲームを初期化し、
```

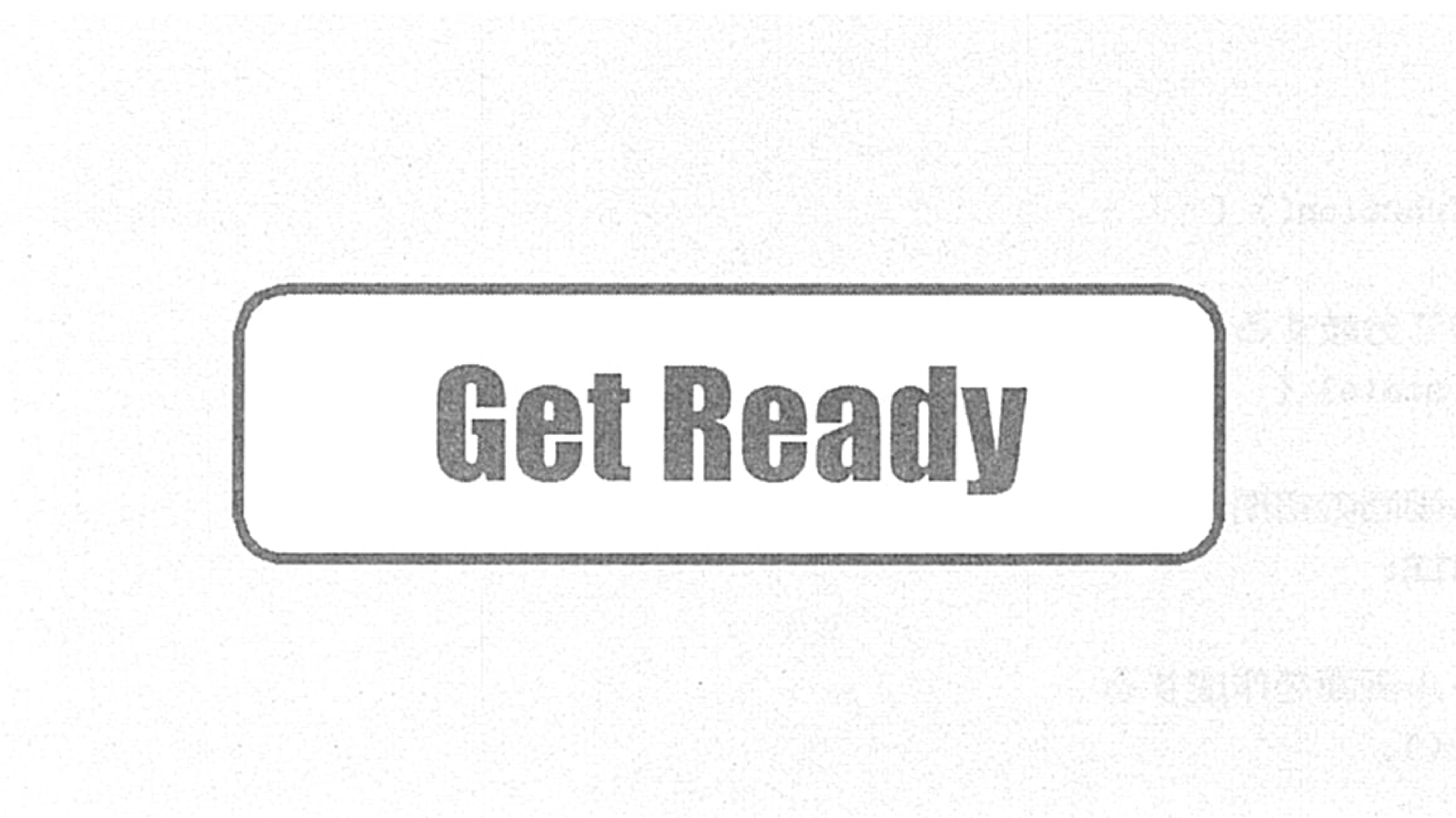
```
        // レディ画面に遷移する
        if (game.isKey(KEY_ENTER)) {
          initGame();
          game.state=GS_READY;
        }
        break;

      // ...(中略)...
    }
  }
}
```

レディ画面

レディ画面は、ゲーム画面にレディ画面用の画像を重ねて表現します(Fig.12)。画像の周囲にある灰色の部分は、アルファ値(p.10)を0にすることで透明にしてあるので、下のゲーム画面が透けて見えます。

▼ **Fig.12**: レディ画面の画像



List3はレディ画面のプログラムです。レディ画面はスプライトを使って表示します。スプライトは、ステージの初期化を行うinitStage関数で、

```
splash=game.createSprite();
```

のように初期化します。splash（スプラッシュ）というのは、ソフトウェアの起動時などに画面の中央に表示する画像のことです。レディ画面も中央に表示するので、ここではsplashという変数名を使いました。

このスプライトについては、

```
splash.setSize(CELL_XCOUNT, CELL_YCOUNT);
```

のようにサイズを画面一杯にします。また、

```
splash.setState(STATE_SPLASH);
```

のように状態を設定します。状態を設定するのは、自機や敵といった他のスプライトと区別するためです。

スプライトの表示はinitSplash関数で行います。ここでは、

```
splash.setXY(game.getViewX(), game.getViewY());
```

として、スクロール座標をスプライトの座標に設定します。これでスクロール座標に関わらず、常にスプライトが画面内に表示されます。次に、

```
splash.initAnim(new Array(image));
```

として、引数imageで指定された画像ファイル名だけの配列を作り、アニメーションとして設定します。これで指定された画像が表示されます。

レディ画面に特有の処理は、メインルーチン（main関数）で行います。レディ画面の初期化（GS_READY）では、

```
initSplash("ready.png");
```

のようにレディ画面の画像（ready.png）を表示し、

```
game.time=0;
```

のようにタイマーを初期化してから、

```
game.state=GS_READY2;
```

として、レディ画面の本体に遷移します。

レディ画面の本体(GS_READY2)では、

```
game.time++;
```

のようにタイマーを更新し、

```
if (game.time==20) {
    initSplash("void.png");
    game.state=GS_GAME;
}
```

として、タイマーが一定値(20)になったら、レディ画面を終わります。空の画像(void.png)を表示してレディ画面を消し、ゲーム画面(GS_GAME)に遷移します。

▼ **List3**: レディ画面(gameUI.js)

```
// スプライトの種類
var STATE_SPLASH=5;

// 各種画面の表示に使うスプライト
var splash;

// ステージの初期化
function initStage() {
  // ...(中略)...

  // スプライトの作成
  splash=game.createSprite();

  // サイズと状態の設定
  splash.setSize(CELL_XCOUNT, CELL_YCOUNT);
  splash.setState(STATE_SPLASH);
}

// 各種画面の表示
function initSplash(image) {

  // スプライトが常に画面内に表示されるように、
  // 座標をスクロール座標と同じ値にする
  splash.setXY(game.getViewX(), game.getViewY());
```

```
  // 指定された画像をアニメーションとして設定する
  splash.initAnim(new Array(image));
}

// メインルーチン
function main() {
  // ...(中略)...

  // 更新処理
  game.onUpdate=function() {

    // 画面に応じて分岐する
    switch (game.state) {
      // ...(中略)...

      // レディ画面の初期化
      case GS_READY:

        // 画像の表示
        initSplash("ready.png");

        // タイマーの初期化
        game.time=0;

        // 本体に遷移する
        game.state=GS_READY2;

      // レディ画面の本体
      case GS_READY2:

        // タイマーの更新
        game.time++;

        // タイマーが一定値に達したときの処理
        if (game.time==20) {

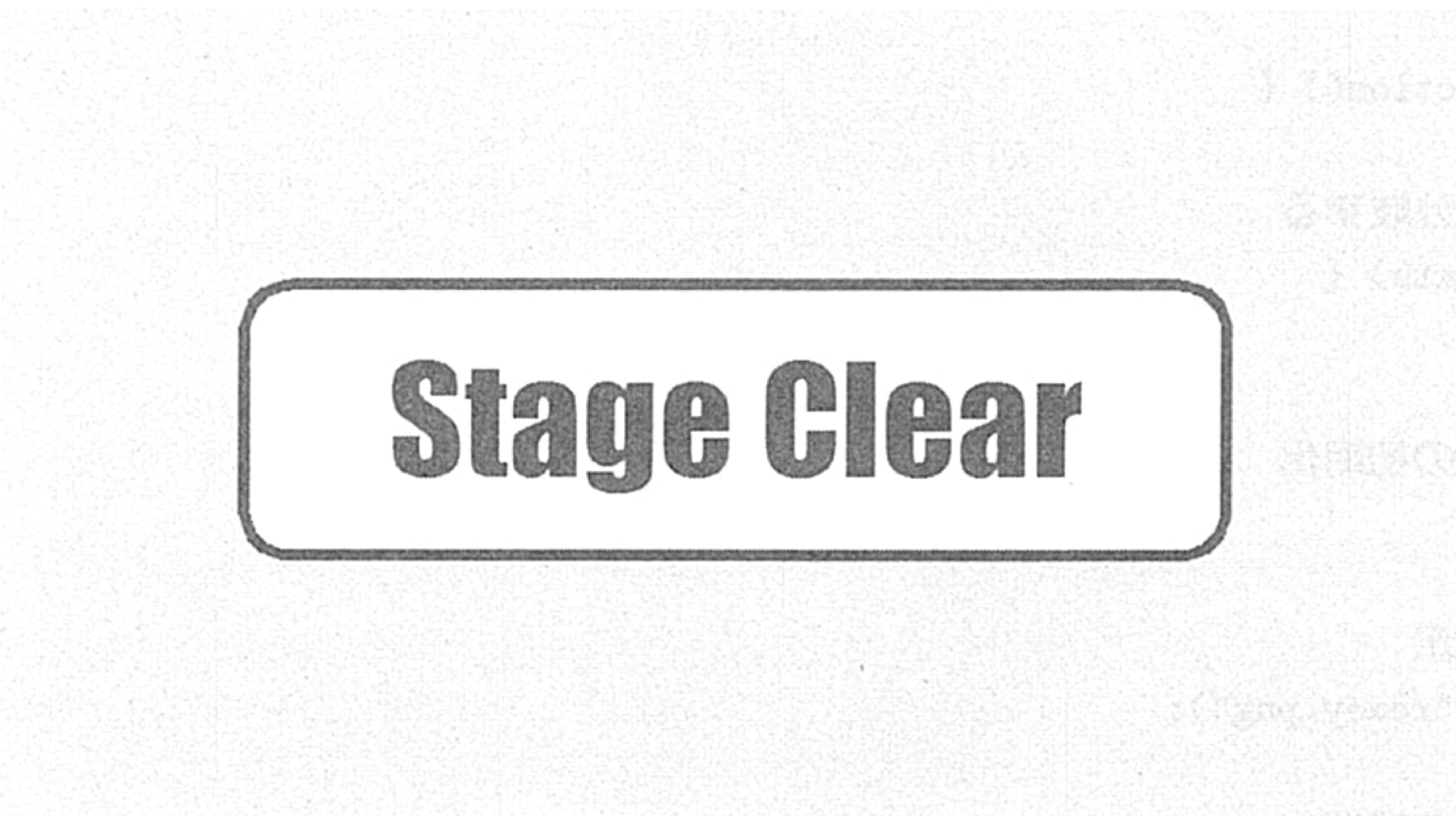
          // 画像の消去
          initSplash("void.png");

          // ゲーム画面に遷移する
          game.state=GS_GAME;
        }
        break;
    }
  }
}
```

クリア画面

クリア画面はレディ画面と同じく、クリア画面用の画像をゲーム画面に重ねて表現します(Fig.13)。灰色の部分は透明です。

▼ **Fig.13**: クリア画面用の画像



List4はクリア画面のプログラムです。レディ画面と同じスプライト(splash)を使って、クリア画面用の画像を表示します。

メインルーチン(main関数)では、レディ画面の初期化(GS_CLEAR)と本体(GS_CLEAR2)の処理を行います。初期化については、

```
initSplash("clear.png");
```

のように画像を表示し、

```
game.time=0;
```

のようにタイマーを初期化してから、

```
game.state=GS_CLEAR2;
```

として、クリア画面の本体に移行します。

本体では、

```
game.time++;
```

のようにタイマーを更新し、

```
if (game.time==20) {
    initSplash("void.png");
    game.addScore(1000);
    initStage();
    game.state=GS_READY;
}
```

として、タイマーが一定値(20)に達したら、クリア画面を消します。さらにスコアに1000点を加算し、新しいステージを初期化してから、レディ画面に遷移します。

一方、通常状態とパワーアップ状態の自機の処理(alive関数)では、

```
if (game.itemCount==0) {
    game.state=GS_CLEAR;
}
```

のような処理を行います。ここでは全てのアイテムを拾ったときに、クリア画面に遷移します。

List4: クリア画面(gameUI.js)

```
// 自機
function Player(x, y) {
  // ...(中略)...

  // 通常状態とパワーアップ状態
  obj.alive=function() {
    // ...(中略)...

    // 自機の座標が整数のとき(座標とセル座標が一致しているとき)
    if (x==cx && y==cy) {

      // エサアイテムに接触したとき
      if (field.getCell(cx, cy)==CELL_ITEM) {

        // エサアイテムを消す(空間のセルにする)
        field.setCell(cx, cy, CELL_VOID);

        // スコアを加算する
        game.addScore(SCORE_ITEM);
```

```
          // 残りのエサアイテム数を減らす
          game.itemCount--;

          // 全てのアイテムを拾ったらクリア画面に移行する
          if (game.itemCount==0) {
            game.state=GS_CLEAR;
          }
        }

        // ...(中略)...
      }
    }
  }

// メインルーチン
function main() {
  // ...(中略)...

  // 更新処理
  game.onUpdate=function() {

    // 画面に応じて分岐する
    switch (game.state) {
      // ...(中略)...

      // クリア画面の初期化
      case GS_CLEAR:

        // 画像の表示
        initSplash("clear.png");

        // タイマーの初期化
        game.time=0;

        // 本体に遷移する
        game.state=GS_CLEAR2;

      // クリア画面の本体
      case GS_CLEAR2:

        // タイマーの更新
        game.time++;

        // タイマーが一定値に達したときの処理
        if (game.time==20) {

          // 画像の消去
```

```
            initSplash("void.png");

            // スコアの加算
            game.addScore(1000);

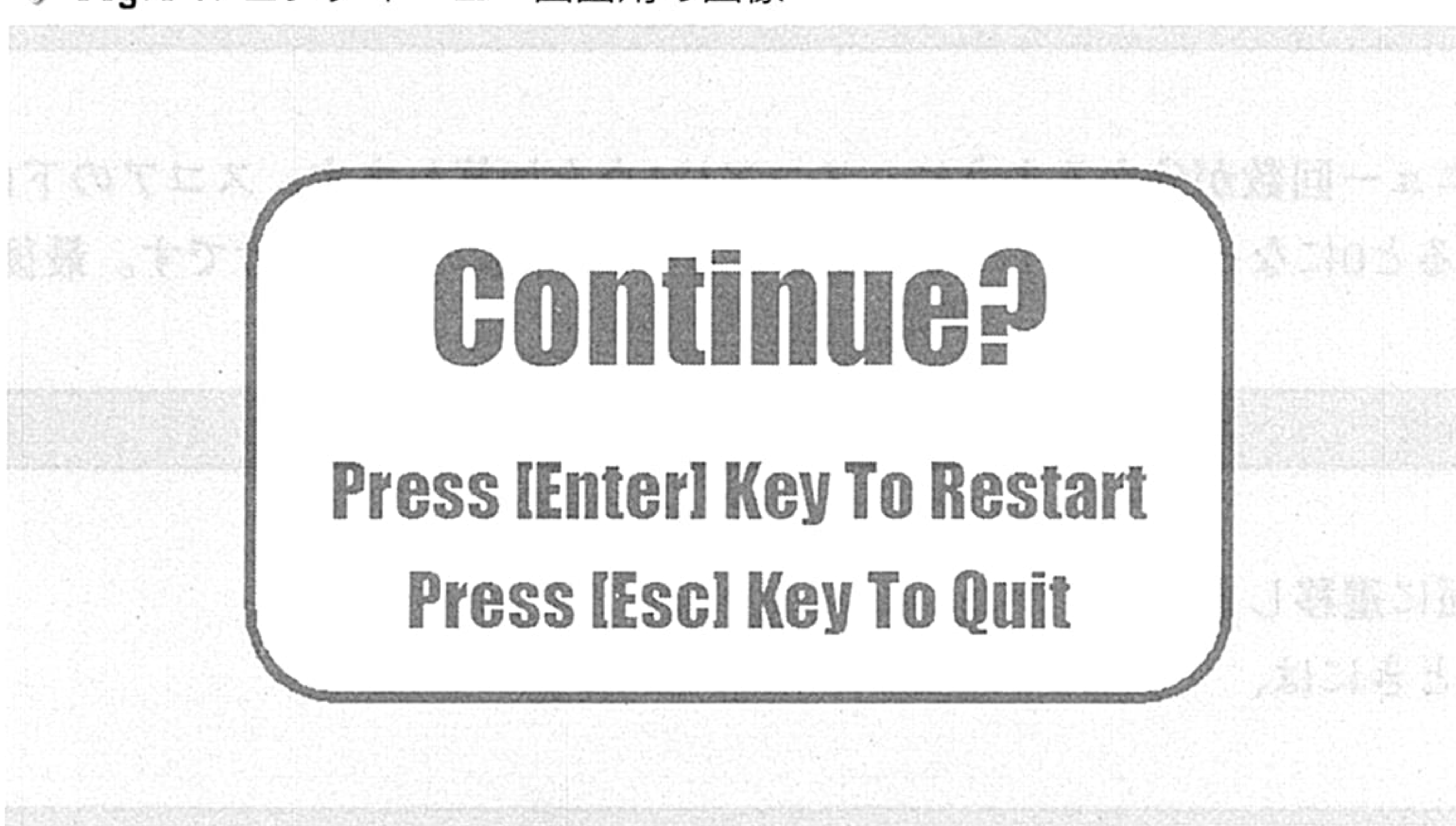
            // ステージの初期化
            initStage();

            // レディ画面に遷移する
            game.state=GS_READY;
          }
          break;
      }
    }
  }
```

コンティニュー画面

コンティニュー画面もレディ画面やクリア画面と同様に、コンティニュー画面用の画像をゲーム画面に重ねて表現します(Fig.14)。灰色の部分は透明です。

▼ **Fig.14**: コンティニュー画面用の画像



List5はコンティニュー画面のプログラムです。メインルーチン(main関数)では、コンティニュー画面の初期化(GS_CONTINUE)と本体(GS_CONTINUE2)の処理を行います。初期化では、

```
initSplash("continue.png");
game.state=GS_CONTINUE2;
```

のように、コンティニュー画面の画像を表示し、本体に遷移します。

本体では、Enterキーを押したらゲームを再開し、Escキーを押したらゲームオーバー画面に遷移します。Enterキーを押したときには、

```
initSplash("void.png");
```

のようにコンティニュー画面を消去します。次に、

```
player.init();
for (i=0; i<enemy.length; i++) {
    enemy[i].init();
}
```

として、自機と敵を初期化します。続いて、

```
if (game.getScore()%10<9) {
    game.addScore(1);
}
```

として、コンティニュー回数が分かるように、スコアに1点を加算します。スコアの下1桁が9のときに1を加算すると0になってしまうので、加算するのは9未満のときだけです。最後に、

```
game.state=GS_READY;
```

として、レディ画面に遷移します。

Escキーを押したときには、

```
initSplash("void.png");
game.state=GS_GAMEOVER;
```

のように、コンティニュー画面を消去して、ゲームオーバー画面に遷移します。

消滅状態の自機の処理(dead関数)では、

```
if (obj.sprite.isAnimOver()) {
    game.state=GS_CONTINUE;
}
```

のような処理を行います。ここでは自機が消滅したときに、コンティニュー画面に遷移します。

List5: コンティニュー画面(gameUI.js)

```
// 自機
function Player(x, y) {
  // ...(中略)...

  // 消滅状態
  obj.dead=function() {

    // アニメーションを再生する
    obj.sprite.stepAnim();

    // アニメーションが終わったらコンティニュー画面に移行する
    if (obj.sprite.isAnimOver()) {
      game.state=GS_CONTINUE;
    }
  }
}

// メインルーチン
function main() {
  // ...(中略)...

  // 更新処理
  game.onUpdate=function() {

    // 画面に応じて分岐する
    switch (game.state) {
      // ...(中略)...

      // コンティニュー画面の初期化
      case GS_CONTINUE:

        // 画像の表示
        initSplash("continue.png");

        // 本体に遷移する
        game.state=GS_CONTINUE2;

      // コンティニュー画面の本体
```

```
      case GS_CONTINUE2:

        // Enterキーを押したとき
        if (game.isKey(KEY_ENTER)) {

          // 画像の消去
          initSplash("void.png");

          // 自機と敵の初期化
          player.init();
          for (i=0; i<enemy.length; i++) {
            enemy[i].init();
          }

          // スコアに1点を加算する
          if (game.getScore()%10<9) {
            game.addScore(1);
          }

          // レディ画面に遷移する
          game.state=GS_READY;
        }

        // Escキーを押したとき
        if (game.isKey(KEY_ESC)) {

          // 画像の消去
          initSplash("void.png");

          // ゲームオーバー画面に遷移する
          game.state=GS_GAMEOVER;
        }
        break;
    }
  }
}
```

ゲームオーバー画面

ゲームオーバー画面も他の画面と同様に、ゲームオーバー画面用の画像をゲーム画面に重ねて表現します(Fig.15)。灰色の部分は透明です。

▼ Fig.15: ゲームオーバー画面用の画像



List6はゲームオーバー画面のプログラムです。メインルーチン(main関数)では、ゲームオーバー画面の初期化(GS_GAMEOVER)と本体(GS_GAMEOVER2)の処理を行います。

初期化では、

```
initSplash("gameover.png");
game.time=0;
game.state=GS_GAMEOVER2;
```

のように、画像を表示し、タイマーを初期化して、本体に遷移します。

本体では、

```
game.time++;
```

のようにタイマーを更新し、

```
if (game.time==20) {
    initSplash("void.png");
    game.state=GS_TITLE;
}
```

として、タイマーが一定値(20)になったら、ゲームオーバー画面を消去し、タイトル画面に遷移します。

▼ **List6**: ゲームオーバー画面(gameUI.js)

```
// メインルーチン
function main() {
  // ...(中略)...

  // 更新処理
  game.onUpdate=function() {

    // 画面に応じて分岐する
    switch (game.state) {
      // ...(中略)...

      // ゲームオーバー画面の初期化
      case GS_GAMEOVER:

        // 画像の表示
        initSplash("gameover.png");

        // タイマーの初期化
        game.time=0;

        // 本体に遷移する
        game.state=GS_GAMEOVER2;

      // ゲームオーバー画面の本体
      case GS_GAMEOVER2:

        // タイマーの更新
        game.time++;

        // タイマーが一定値に達したときの処理
        if (game.time==20) {

          // 画像の消去
          initSplash("void.png");

          // タイトル画面に遷移する
          game.state=GS_TITLE;
        }
        break;
    }
  }
}
```

Chapter 11

ネットワーク

ネットワークを使ったランキング機能

ゲームはスコアが全てではありませんが、より高いスコアを目指すのは楽しい遊び方の一つです。特に他の人とハイスコアを競うのは、一人でハイスコアを目指す以上の刺激があります。

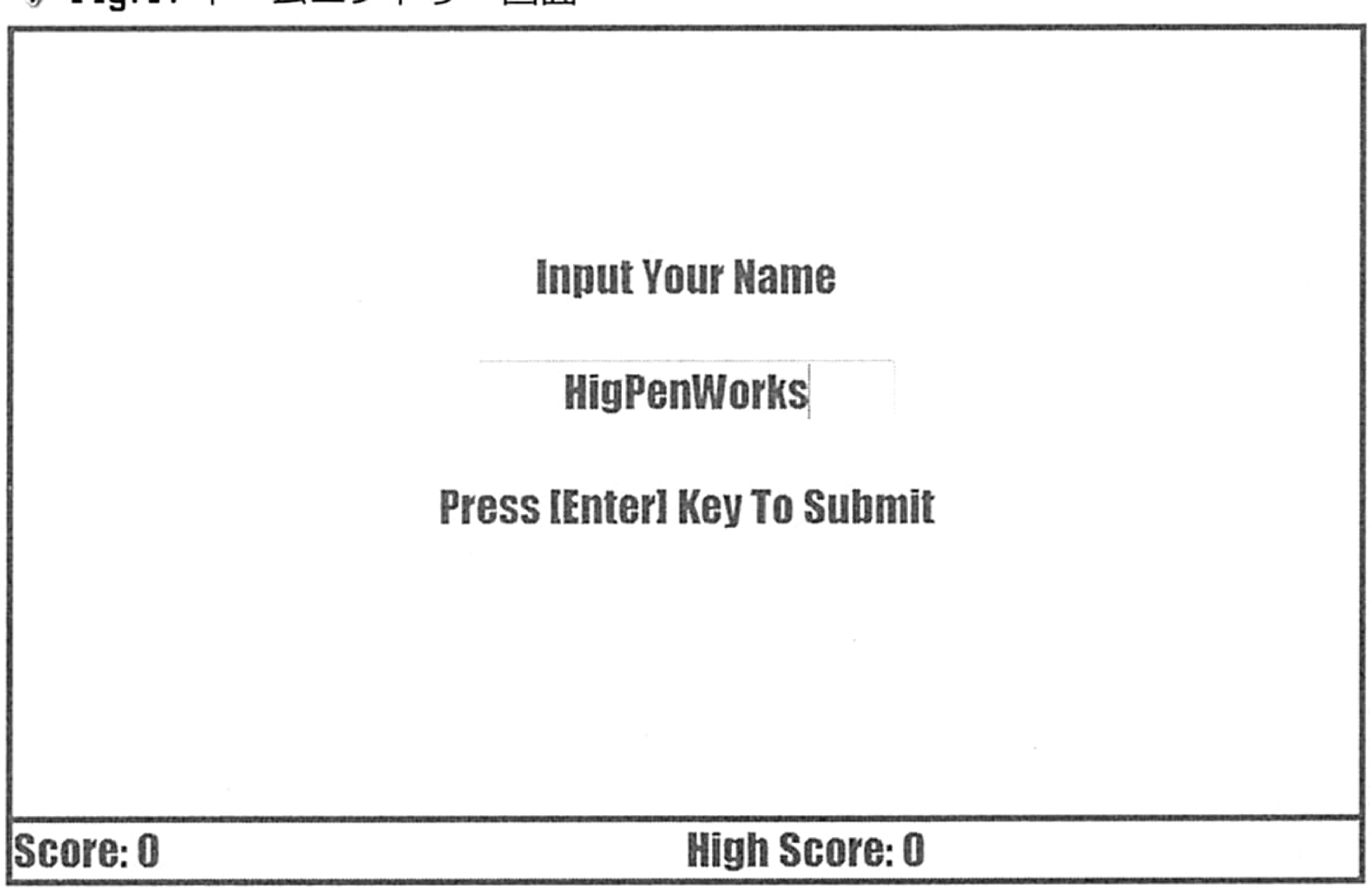
ハイスコアを競う手段として、ネットワークを使ったランキング機能が今や当たり前になりました。ハイスコアを出すと、ゲームプログラムがネットワーク上のサーバへ自動的にアクセスして、スコアや名前を登録します。また、ハイスコア情報をサーバから取得して、ランキングを画面に表示します。

今までのサンプルにもハイスコアを記録する機能がありましたが、これはネットワークを使ったものではないので、他の人とスコアを見せあうことはできません。またブラウザを閉じると、ハイスコアは消えてしまいます。

そこで本章では、第10章のアイテムを拾うゲーム(p.310)に対して、ネットワークを使ったランキング機能を追加しました。ゲームを実行するには、後述する方法で必要なファイルをサーバにアップロードしてから、ブラウザでranking.htmlを開きます。サーバにアップロードした状態でなければ、このプログラムは正しく動作しません。アップロードの作業を省略して簡単に実行するには、著者のWebサイト(http://cgi32.plala.or.jp/higpen/gate.shtml)で公開しているサンプルをブラウザで開いてください。

ゲームを起動すると、最初にネームエントリー画面が表示されます(Fig.1)。ここではランキングに登録する名前を入力してください。ハイスコアを出すと、ここで入力した名前とスコアがサーバに記録されます。

▼ **Fig.1**: ネームエントリー画面

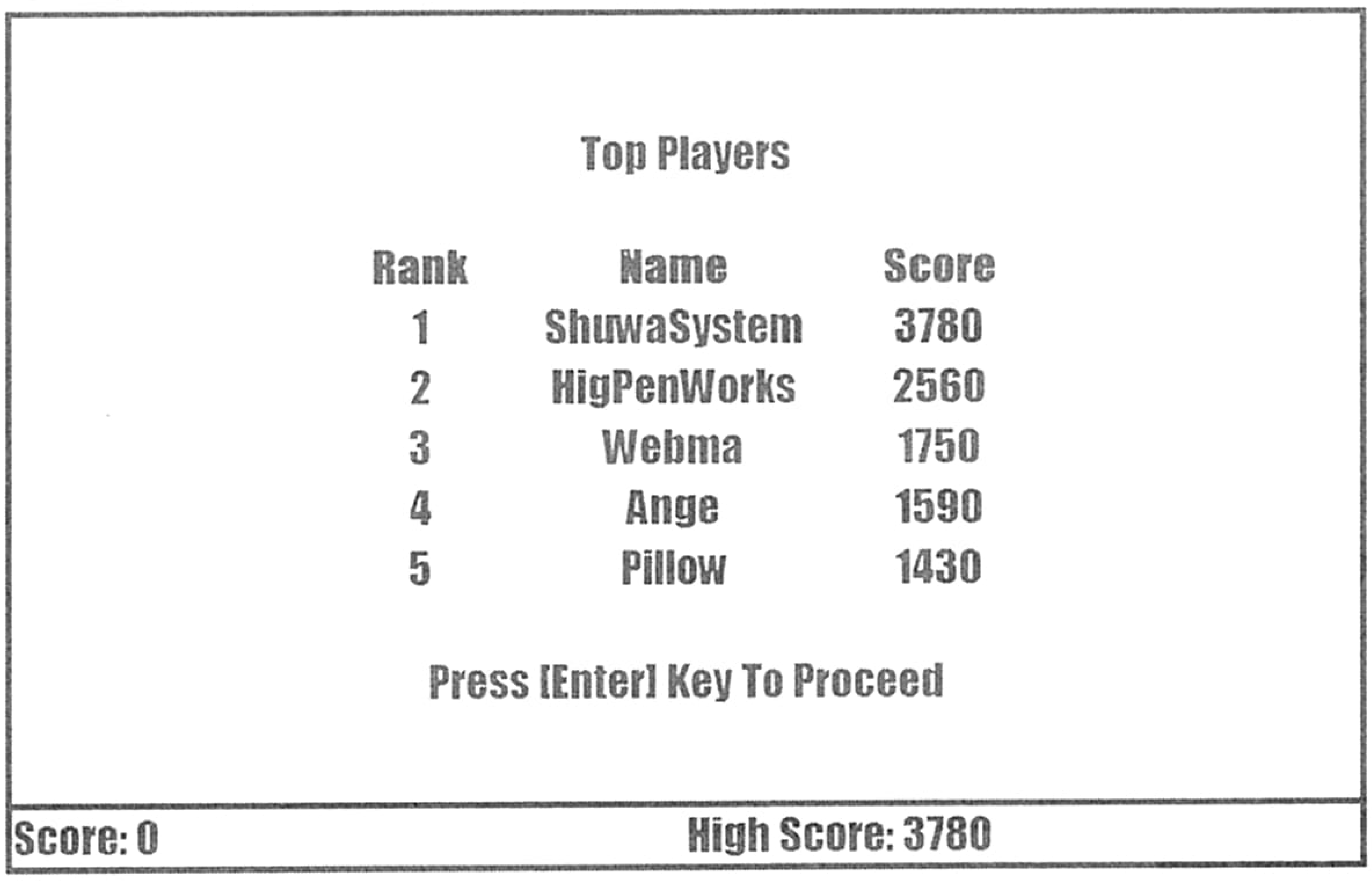


名前には英字(A～Z, a～z)と数字(0～9)が使えます。スペース(空白)や各種の記号、日本語の文字は使えません。

入力した名前を変えたいときには、ゲームを再起動してください。ハイスコアを出すたびに名前を入力するのは面倒なので、このようにゲームの起動時に名前を入力する方式にしました。

名前を入力してEnterキーを押すと、ランキング画面に移ります(Fig.2)。ここではサーバから取得した上位5位のスコアと名前を表示します。

▼ **Fig.2**: ランキング画面

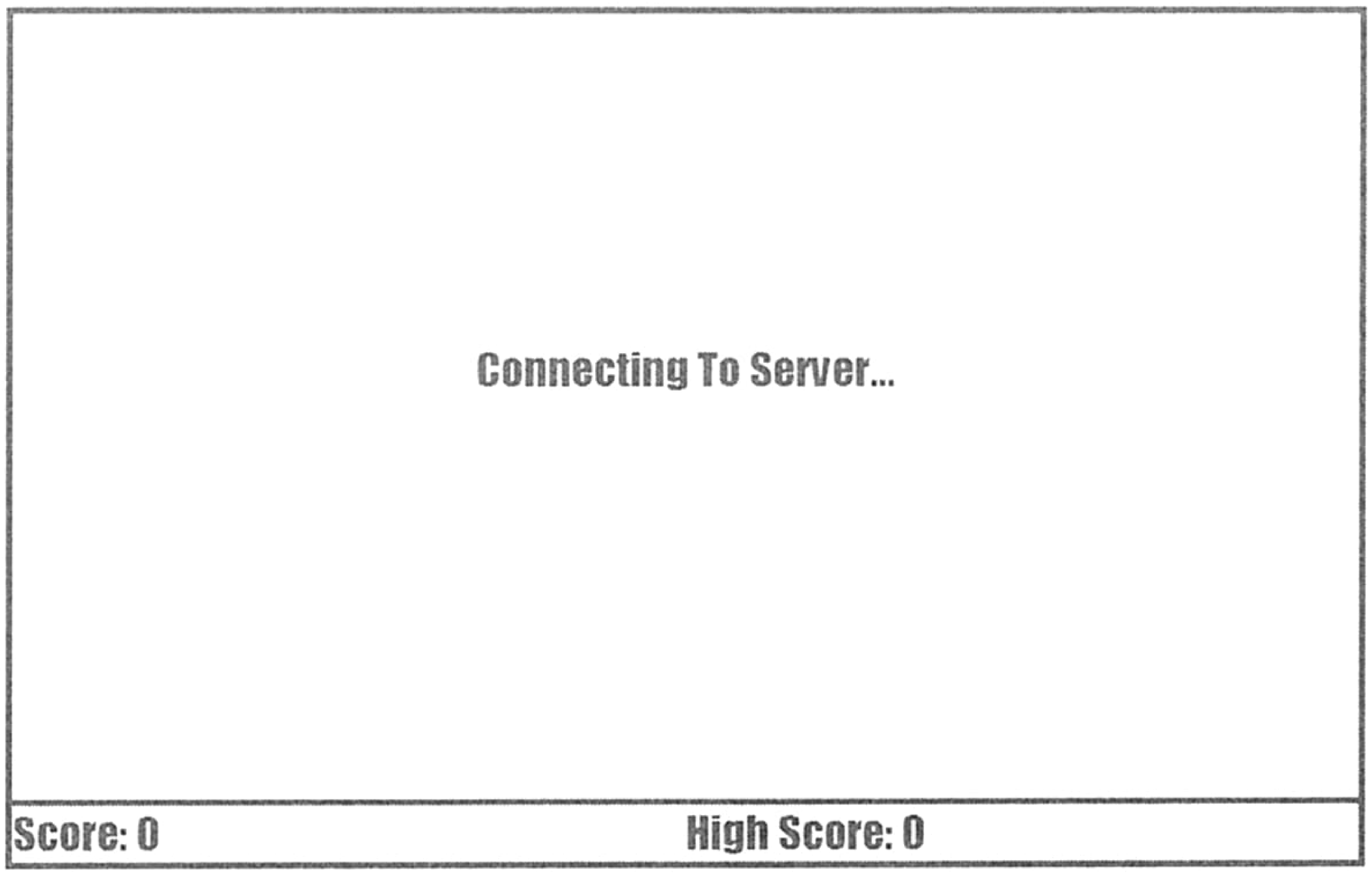


ランキング画面でEnterキーを押すと、タイトル画面に移ります。あとの操作方法は第10章のサンプルと同じです。ゲームオーバー画面の後には再びランキング画面に移り、もしハイスコアを出していたら、スコアと名前が登録されます。

コンティニューをした場合には、コンティニューをする前のスコアだけがハイスコアとして登録されます。例えば1730点でゲーム終了となり、コンティニュー後に2171点を出した場合には、1730点だけがサーバに登録されます。

ランキング画面を表示する際には、サーバとの通信を行います。通信待ちの間には「Connecting To Server...」というメッセージが表示されますが(Fig.3)、通常は一瞬でメッセージが消えて、ランキング画面になります。

▼ Fig.3: 通信待ちのメッセージ

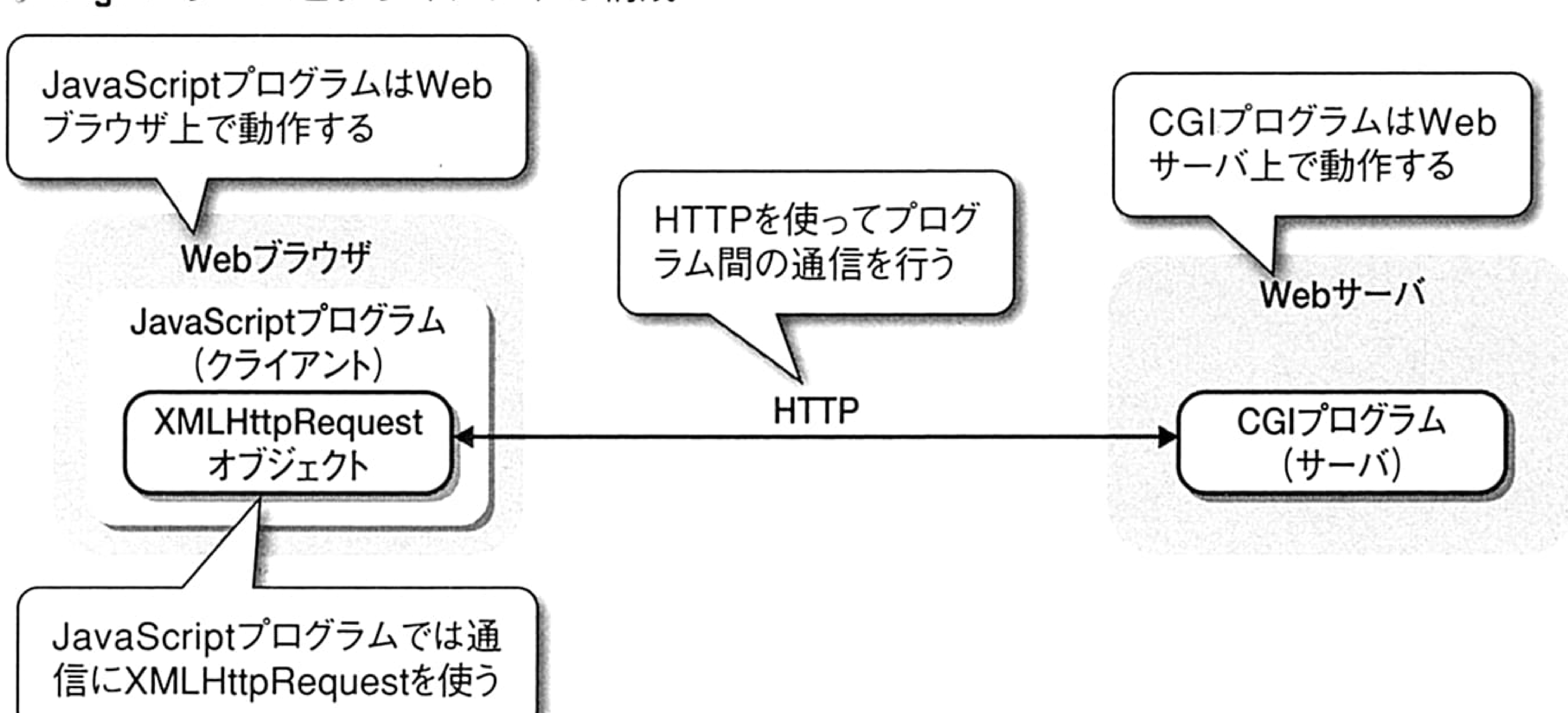


もしメッセージが長時間表示されたままになる場合は、通信エラーが発生している可能性があります。サーバにファイルをアップロードした場合には、ファイルの構成が正しいかどうかを確認してください。

ファイルの構成

サンプルのランキング機能は、サーバ側にPerlによるCGIプログラム、クライアント側にJavaScriptプログラムを使って実現しました。通信にはAjaxで用いられるXMLHttpRequestオブジェクトを使い、HTTPでサーバとクライアントを結びます(Fig.4)。HTTPはWebサーバとWebブラウザの間で用いる通信プロトコルです。

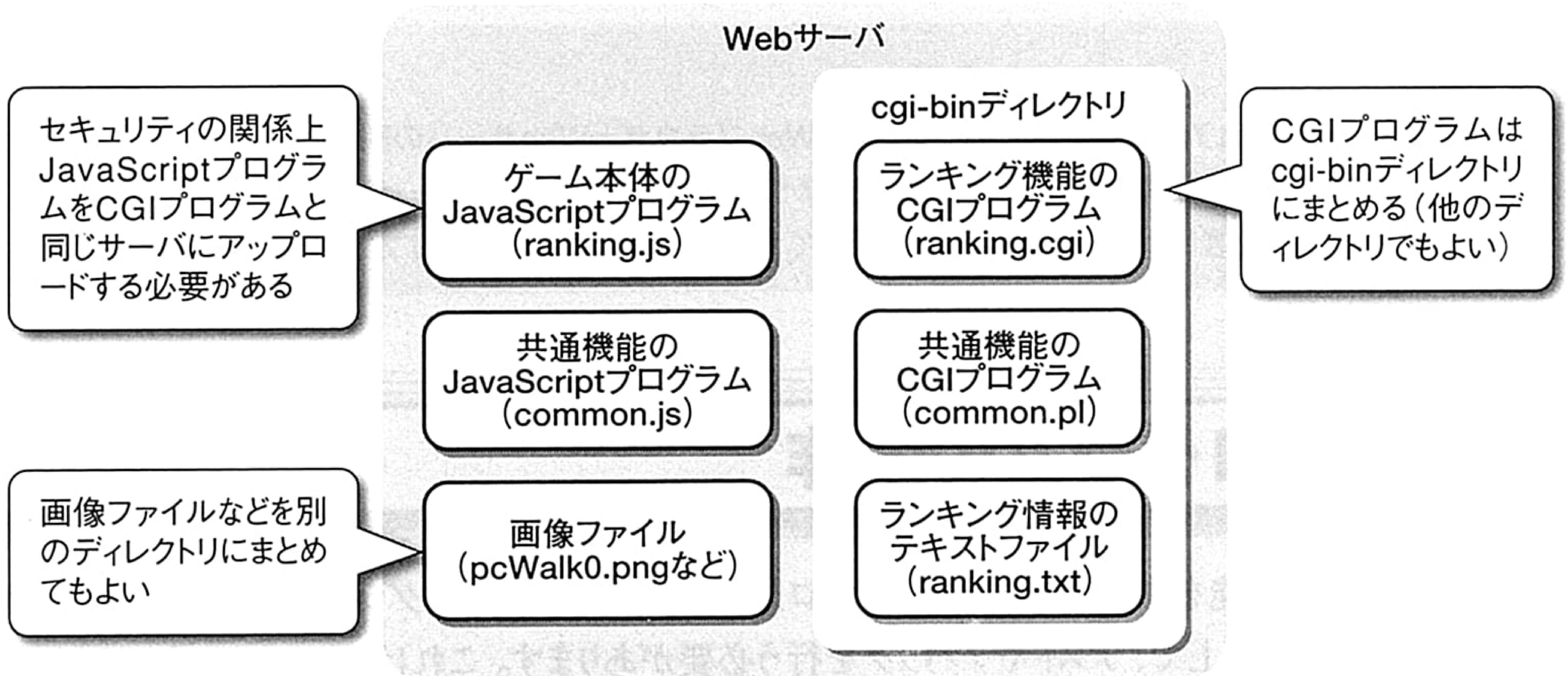
▼ Fig.4: サーバとクライアントの構成



CGIプログラムとJavaScriptプログラムの間で通信を行うには、セキュリティの関係上、JavaScriptプログラムをCGIプログラムと同じサーバ上にアップロードしておく必要があります。JavaScriptプログラムが別のサイトにある場合や、JavaScriptプログラムがローカルマシン(手元のPC)上にある場合には、CGIプログラムとの通信ができません。このような制限があるのは、JavaScriptプログラムがユーザの想定していないサイトと勝手に通信する危険性を回避するためです。

サンプルではJavaScriptプログラムと、JavaScriptプログラムから使う画像ファイルの一式をサーバに配置します(Fig.5)。CGIプログラムとランキングを記録したテキストファイルは、cgi-binディレクトリに配置することにしました。

Fig.5: ファイルの構成



ファイルのパーミッション(アクセス許可)は、

- JavaScriptプログラム、画像ファイル:
  644 (読み取りが可能、所有者は書き込みも可能)
- cgi-binディレクトリ:
  755 (読み取りと実行が可能、所有者は書き込みも可能)
- ランキング機能のCGIプログラム:
  700 (所有者のみ読み取り・書き込み・実行が可能)
- 共通機能のCGIプログラム、ランキングのテキストファイル:
  600 (所有者のみ読み取り・書き込みが可能)

のように設定します。パーミッションはほとんどのFTPクライアントソフトウェアで設定できるので、ファイルをアップロードしたときに設定してください。

動作テストは一般のプロバイダで行いました。PerlによるCGIプログラムが動作する環境ならば、プロバイダでも自前のサーバでも、同様に動作するはずです。テスト環境のPerlのバージョンは5.8.3です。

Ajax（エイジャックス、アジャックス）

Ajax（Asynchronous JavaScript + XML）は、WebブラウザとWebサーバの間で非同期に通信を行い、さまざまなサービスを実現する技術の総称です。JavaScriptのXMLHttpRequestオブジェクトを用いて、Webサーバとの間で任意のタイミングで通信を行うことにより、従来のWebよりも柔軟性の高い機能を提供することができます。Ajaxが指す技術の範囲は広く、例えば本書のようにXMLは使わずにXMLHttpRequestオブジェクトだけを使う場合も、Ajaxと呼ぶことが多いようです。

HTTP

HTTP（Hypertext Transfer Protocol）は、主にWebブラウザとWebサーバの間でデータを送受信するために使う通信プロトコルです。クライアント（Webブラウザ）がサーバ（Webサーバ）に対してGETやPOSTといったリクエストを送信し、このリクエストに対してサーバがレスポンスを返す方式で通信を行います。

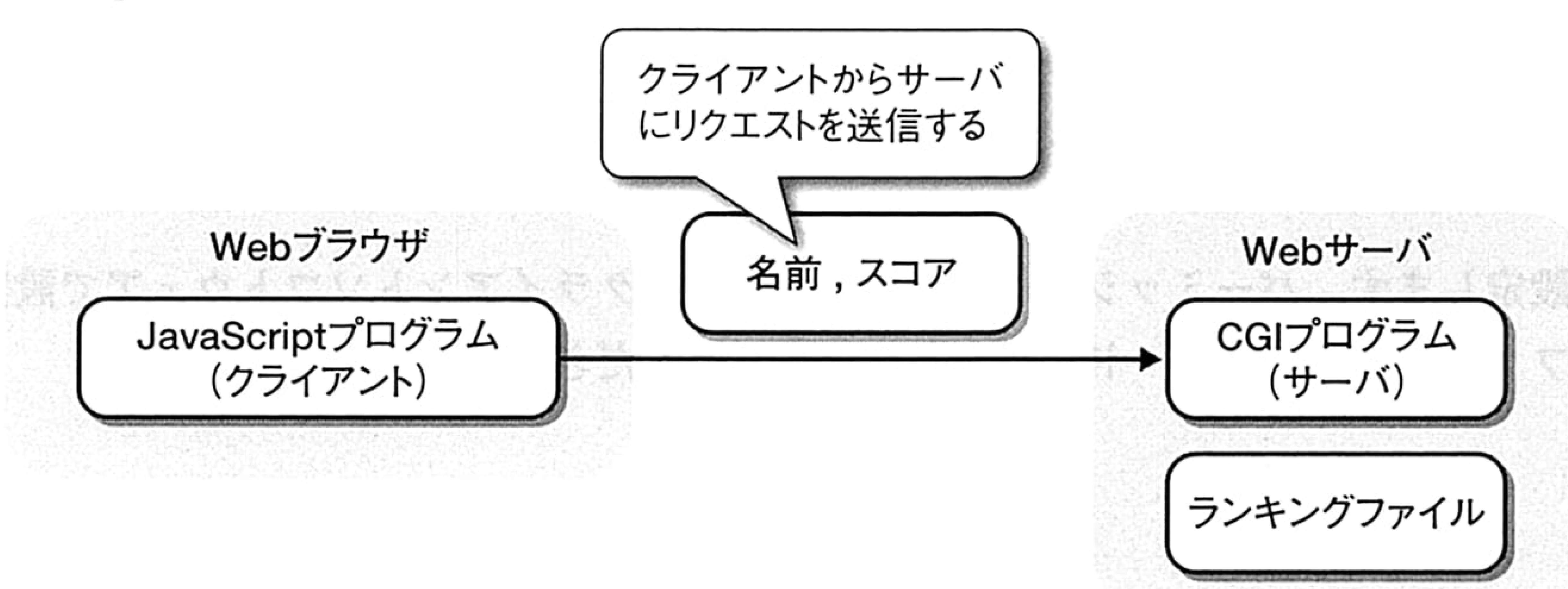
CGIプログラムの制作

ランキング機能を実現するには、CGIプログラムとJavaScriptプログラムの両方を作成し、サーバにアップロードした上で、テストやデバッグを行う必要があります。これはJavaScriptプログラムだけを作ることに比べると非常に難しそうですが、制作の順番を工夫すれば意外に大変ではありません。

最初はCGIプログラムから作るとよいでしょう。CGIプログラムは単体でもテストができるからです。CGIプログラムが正しく動いたところで、JavaScriptプログラムの制作に取りかかるのがお勧めです。

ランキング機能のためのCGIプログラムは、次のような処理を行います。まずJavaScriptプログラムから、名前とスコアの組が送信されます（Fig.6）。これをリクエストと呼びます。リクエストというのは、HTTPにおいてクライアントからサーバに対して送信するメッセージのことです。CGIプログラムはリクエストを解釈して、名前とスコアを取得します。

Fig.6: リクエストの受信



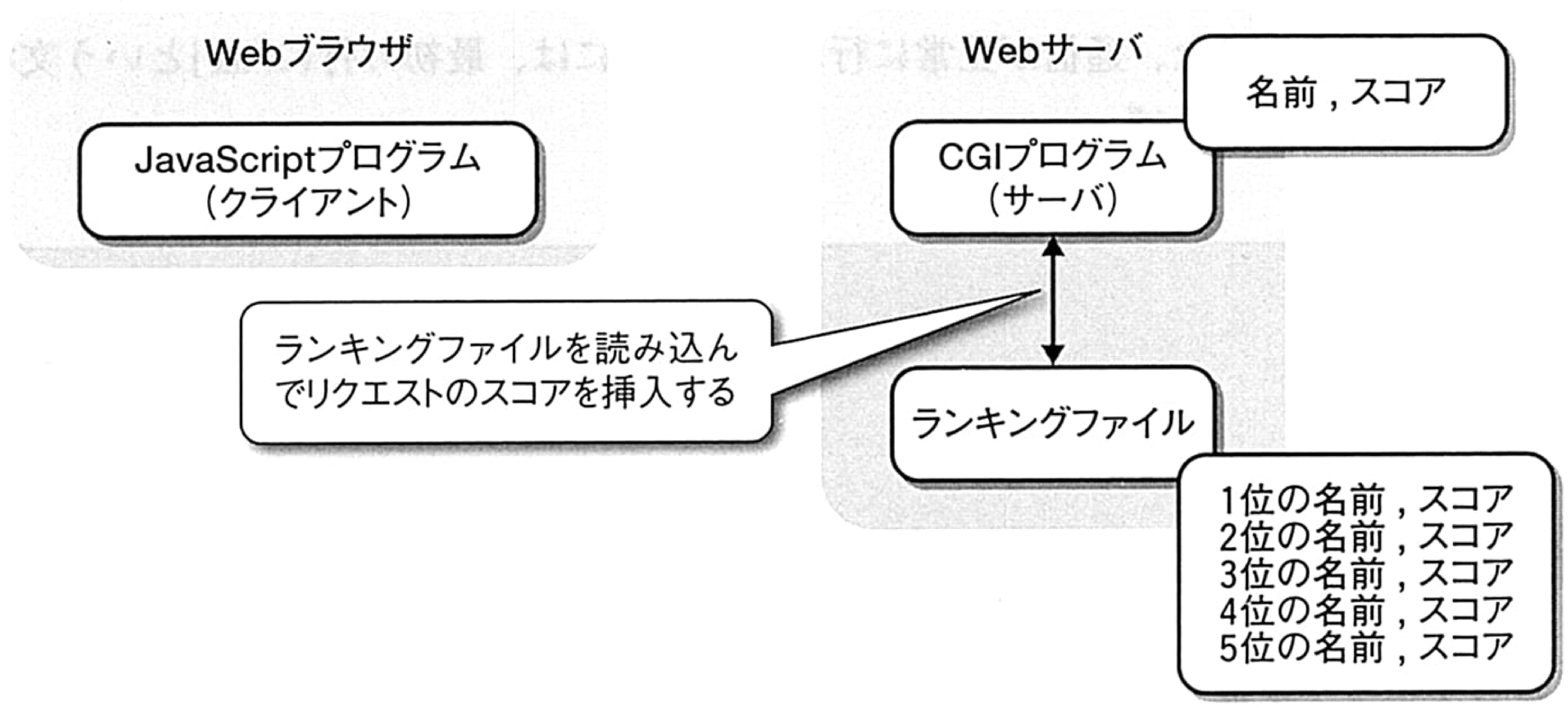
リクエストの形式は自由に決めて構いません。サンプルのリクエストでは、名前とスコアをカンマ(,)で区切って並べることにしました。例えば、

```
HigPenWorks,2560
```

のような文字列を、リクエストとして送信します。

次に、サーバ上にあるランキングファイル(ランキングを記録したテキストファイル)を読み込んで、リクエストのスコアと比べます(Fig.7)。リクエストのスコアがランキングに入るならば、ランキングの適切な位置に名前とスコアを挿入します。ランキングに入らなければ、スコアは登録しません。登録が終わったら、ランキングファイルを出力します。

▼ **Fig.7**: スコアと名前の登録



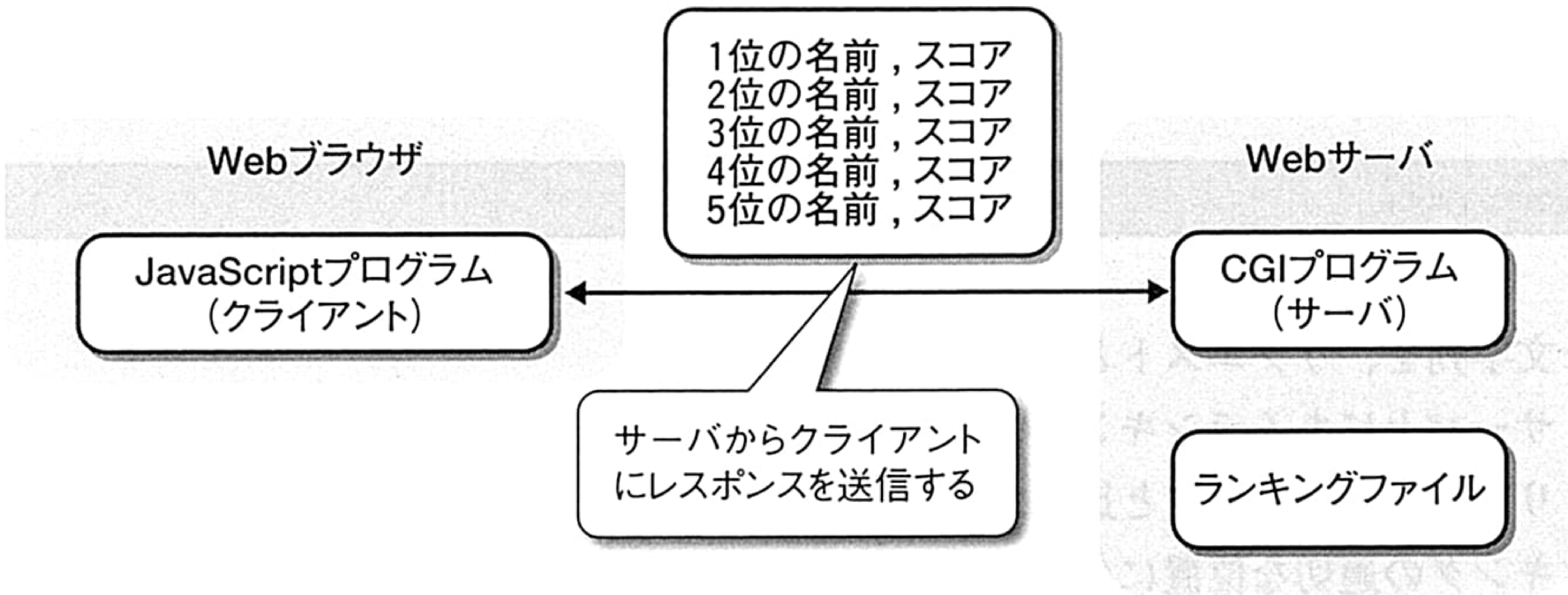
ランキングファイルの形式は自由に決めて構いません。サンプルでは、

```
ShuwaSystem,3780
HigPenWorks,2560
Webma,1750
Ange,1590
Pillow,1430
```

のように、名前とスコアをカンマ(,)で区切った組を、改行で区切って並べることにしました。ランキングは5個まで記録して、6個目以降は削除することにします。

最後に、JavaScriptプログラムに対してランキング情報を送信します(Fig.8)。これをレスポンスと呼びます。レスポンスというのは、HTTPにおいてサーバからクライアントに対して送信するメッセージのことです。HTTPでは、クライアントがサーバに対してリクエスト(要求)を送信し、サーバがそれに対するレスポンス(返答)を送信するという形式で、情報をやりとりします。

▼ Fig.8：レスポンスの送信



レスポンスの形式も、アプリケーションの都合により自由に決めて構いません。サンプルのレスポンスでは、名前とスコアをカンマ(,)で区切ったものを、さらに改行で区切って複数並べることにしました。また、通信が正常に行われたときには、最初の行に「ok」という文字列を入れることにします。例えば、

```
ok
ShuwaSystem,3780
HigPenWorks,2560
Webma,1750
Ange,1590
Pillow,1430
```

のようなレスポンスを返します。これはランキングファイルの冒頭に「ok」の行を追加した形式になっています。一方、リクエストが適切な形式でないときや、ランキングファイルにアクセスできないときなどは、

```
error,parameter
```

や、

```
error,file
```

のように、エラーが起きたことを示すレスポンスを返します。

このようにリクエストとレスポンスを分かりやすく設計しておくと、プログラムが作りやすくなります。一方、CGIプログラムにはJavaScriptプログラム以外からもアクセスできるので、不正にスコアを登録することも可能です。こういった不正アクセスを完全に防止するのは難しい

のですが、リクエストやレスポンスを暗号化したり、CGIプログラムやJavaScriptプログラムを解析しにくくしたりといった対策を行うことによって、不正アクセスを減らすことができます。

通信関連のCGIプログラム

List1は通信関連のCGIプログラムです。CGIプログラムは、ランキング機能の本体(ranking.cgi)と、通信やファイルの機能をまとめたプログラム(common.pl)に分かれています。List1はcommon.plの一部を抜粋したものです。

List1のget_parameterサブルーチンは、JavaScriptプログラムから送信されたリクエストを解釈して、パラメータを取り出します。まず、

```
if (
    $ENV{'REQUEST_METHOD'} ne 'POST' ||
    $ENV{'CONTENT_LENGTH'}>$MAX_LENGTH
) {
```

のように、リクエストの種類がPOSTかどうかと、パラメータの長さが規定値を超えていないかどうかを調べます。POSTでない場合と、パラメータが長すぎる場合には、エラーを出力して終了します。

$ENVは環境変数を表します。例えば、

```
$ENV{'REQUEST_METHOD'}
```

は、環境変数REQUEST_METHODを指します。CGIプログラムは環境変数を介して、リクエストに関するさまざまな情報を取得します。

HTTPのリクエストにはGETやPOSTといったいくつかの種類があります。GETリクエストはサーバに対して情報を要求する際に使います。POSTリクエストはサーバに対して情報を送信し、その結果を受信したいときに使います。今回のランキング機能では、サーバにスコアや名前を送信してランキング情報を取得するので、POSTリクエストを使うことにします。

POSTリクエストで送信された情報をCGIプログラムで取得するには、

```
read(STDIN, $s, $ENV{'CONTENT_LENGTH'});
```

のように、標準入力(STDIN)を使います。ここでは環境変数CONTENT_LENGTHが表す文字数の文字列を標準入力から取得し、変数$sに格納します。

次に、取得したパラメータの中に不適切な文字が含まれていないかどうかを調べます。この

チェックを行わないと、悪意のある利用者が不適切な文字の混入したデータを送信して、ファイルなどを破壊する可能性があります。文字をチェックするには、

```
if ($s!~/^[A-Za-z0-9,]*$/) {
```

のようなif文を使います。!~は正規表現にマッチしなかったかどうかを調べる演算子です。ここでは、

- 英字(A～Z, a-z)
- 数字(0～9)
- カンマ(,)

といった文字以外の文字が文字列に含まれていたときに、エラーを出力して終了します。このサンプルのパラメータは、

```
HigPenWorks,2560
```

のように、

```
名前,スコア
```

という形式です。名前は英字と数字、スコアは数字だけなので、それ以外の文字が含まれていたときはエラーにします。

最後の、

```
return split(',', $s);
```

では、split関数を使って文字列をカンマ(,)で分割し、配列として返します。例えば、

```
ShuwaSystem,3780
```

のようなパラメータは、最初の要素が「ShuwaSystem」で、次の要素が「3780」の配列になります。

▼ **List1**：通信関連のCGIプログラム（common.pl）

```
# リクエストパラメータの最大文字数
$MAX_LENGTH=1000;

# リクエストパラメータの取得
sub get_parameter {

  # サブルーチン内で使うローカル変数
  local($s, $c, $i);

  # POSTかどうか、パラメータ長は適切かどうかを確認する
  if (
    $ENV{'REQUEST_METHOD'} ne 'POST' ||
    $ENV{'CONTENT_LENGTH'}>$MAX_LENGTH
  ) {
    # 不適切な場合にはエラーを出力して終了する
    print "error,parameter";
    exit;
  }

  # パラメータの取得
  read(STDIN, $s, $ENV{'CONTENT_LENGTH'});

  # パラメータに不適切な文字がないかどうかを確認する
  if ($s!~/^[A-Za-z0-9,]*$/) {

    # 不適切な文字がある場合にはエラーを出力して終了する
    print "error,parameter";
    exit;
  }

  # パラメータをカンマ(,)で分割し、配列として返す
  return split(',', $s);
}
```

標準入力

プログラムに対してデータを入力するための仕組みです。特別な入力元を指定しないときに使われる入力元なので、標準入力と呼びます。キーボード・リダイレクト・パイプなどの入力は標準入力になります。

正規表現

さまざまな文字列の集合を、特殊な表記の文字列で表現する方法の一つです。主にプログラム内の文字列処理や、テキストエディタやワードプロセッサなどの検索機能で使われています。正規表現はgrep、awk、vi、Perlといった UNIX系のツールや言語を中心に、さまざまなソフトウェアがサポートしています。

ファイル関連のCGIプログラム

List2はファイル関連のCGIプログラムです。common.plからファイル処理に関する部分を抜粋しました。

まず、ファイルのロックに関するサブルーチンとして、lockexとunlockを用意しました。CGIプログラムがファイルを読み書きするときには、処理の途中で他のCGIプログラムが同じファイルをアクセスしないように、ファイルをロックする必要があります。

lockexはファイルのロック(排他ロック)を行い、unlockはアンロック(ロックの解除)を行います。排他ロックというのは、他のプログラムのアクセスを一切禁止するロックです。ファイルを変更せずに読み込むだけの場合には、排他ロックではなく共有ロックを使うと、他のプログラムも同時にファイルを読み込むことができます。

次のopen_fileサブルーチンは、引数で指定したファイルをロックしてから、内容を配列に読み込みます。ファイルを開くには、

```
if (open(FP, "<$file")==0) {
```

のようにopen関数を使います。FPはファイルポインタと呼ばれ、開いたファイルの内容を読み書きしたり、ファイルを閉じたりする際に利用します。

ファイルを読み込むには、

```
while (eof(FP)!=1) {
```

のようにwhile文とeof関数を使います。ファイルの最後に達するとeof関数は1を返すので、while文を抜け出して、読み込みを終了します。

while文の内部では、

```
$s=<FP>;
```

のようにファイルを1行ずつ変数$sに読み込み、

```
chomp($s);
```

のようにchomp関数で改行文字を除去した後に、

```
push(@a, $s);
```

として、読み込んだ行を配列@aの末尾に追加します。これでファイルを行単位に分割して配列へ読み込むことができます。

open_fileサブルーチンの最後では、

```
close(FP);
```

のようにファイルを閉じてから、

```
return @a;
```

として、ファイルの内容を格納した配列を返します。

open_fileサブルーチンと対になる処理として、close_fileサブルーチンを用意しました。このサブルーチンは、指定した配列の内容をファイルに書き込み、ロックを解除します。

まず、

```
if (open(FP, ">$file")==0) {
```

のようにファイルを開きます。次に、

```
for ($i=0; $i<=$#_; $i++) {
    print FP "$_[$i]\n";
}
```

のようなfor文を使って、配列を要素ごとに改行で区切り、ファイルに書き込みます。ここで、

```
$_[$i]
```

というのは、サブルーチンの引数(`$_`)の`$i`番目の要素を表します。Perlでは、サブルーチンの引数は配列として扱われます。一方、

```
$#_
```

は、サブルーチンの引数の個数(配列の要素数)を表します。

close_fileサブルーチンの最後では、

```
close(FP);
unlock(FP);
```

のようにファイルを閉じて、ロックを解除します。

List2: ファイル関連のCGIプログラム(common.pl)

```
# ファイルの排他ロックを行う
sub lockex {
  flock($_[0], 2);
}

# ファイルのアンロックを行う
sub unlock {
  flock($_[0], 8);
}

# ファイルを開いて読み込む
sub open_file {

  # サブルーチン内で使うローカル変数
  local($s, @a);

  # サブルーチンの第1引数をファイル名とする
  $file=$_[0];

  # ファイルを読み込みモードで開く
  if (open(FP, "<$file")==0) {

    # ファイルが開けなければエラーを出力して終了する
    print "error,file";
    exit;
  }

  # ファイルをロックする
  lockex(FP);

  # ファイルを配列に読み込む
  while (eof(FP)!=1) {

    # ファイルを1行読み込む
    $s=<FP>;
```

```
    # 改行文字を除去する
    chomp($s);

    # 行を配列に格納する
    push(@a, $s);
  }

  # ファイルを閉じる
  close(FP);

  # ファイルを読み込んだ配列を返す
  return @a;
}

# ファイルに書き込んで閉じる
sub close_file {

  # サブルーチン内で使うローカル変数
  local($i);

  # ファイルを書き込みモードで開く
  if (open(FP, ">$file")==0) {

    # ファイルが開けなければエラーを出力する
    print "error,file";

    # ファイルをアンロックして終了する
    unlock(FP);
    exit;
  }

  # 配列をファイルに書き込む
  for ($i=0; $i<=$#_; $i++) {

    # 配列の要素を改行で区切って出力する
    print FP "$_[$i]\n";
  }

  # ファイルを閉じてアンロックする
  close(FP);
  unlock(FP);
}
```

CGIプログラムの本体

List3はランキング機能を提供するCGIプログラムの本体です。最初の、

```
#!/usr/local/bin/perl
```

は、このCGIプログラムを実行するPerlコマンドのパスを表します。パスは動作環境によって異なる可能性があるので、プロバイダの説明などを参考に変更してください。

次に、

```
print "Content-type: text/plain\n\n";
```

のようにレスポンスのヘッダを出力します。これはContent-typeエンティティヘッダフィールドと呼ばれ、ドキュメントの種類を表します。JavaScriptプログラムと通信する場合、Content-typeには何を指定しても構わないのですが、特に理由がなければtext/plainかtext/htmlを記述するとよいでしょう。これはブラウザなどを使ってCGIプログラムをテストする際に便利だからです。

続いて、

```
@param=get_parameter();
```

のように、get_parameterサブルーチン（p.343）を使ってリクエストパラメータを取得し、

```
$new_name=shift(@param);
$new_score=shift(@param);
```

として、名前とスコアを取り出します。このスコアがランキングのスコアを上回っていたら、名前とスコアを登録します。shiftは配列の先頭要素を取り出す関数です。

ランキングファイルは、

```
@line=open_file($RANKING_FILE);
```

のように、open_fileサブルーチン（p.346）を使って読み込みます。そして、

```
for ($i=0; $i<=$#line; $i++) {
```

のようなfor文を使ってランキングファイルの各行を調べ、

```
@a=split(',', $line[$i]);
$name=shift(@a);
$score=shift(@a);
```

のように名前とスコアを取り出します。続いて、

```
if ($score<$new_score) {
```

のように既存のスコアを新しいスコアが上回ったかどうかを調べ、上回っていたら、

```
splice(@line, $i, 0, "$new_name,$new_score");
```

のように新しい名前とスコアを挿入します。spliceは配列に要素を挿入したり、要素を削除したりするための関数です。ここでは配列@lineの$i番目の要素として、名前とスコアをカンマ(,)で区切った文字列を挿入します。

登録が終わったら、

```
splice(@line, $RANKING_COUNT);
```

のようにsplice関数を使って、スコアを上位$RANKING_COUNT個だけ残し、以降のスコアを削除します。そして、

```
close_file(@line);
```

のように、close_fileサブルーチン(p.347)を使って配列をファイルに書き込みます。

最後に、正常に処理が終わったことを示すために、

```
print "ok\n";
```

のように「ok」というレスポンスを返します。さらに、

```
for ($i=0; $i<=$#line; $i++) {
    print "$line[$i]\n";
}
```

のようなfor文を使って、ランキングファイルの内容をレスポンスとして返します。

▼ **List3**: CGIプログラムの本体(ranking.cgi)

```
# Perlコマンドのパス(動作環境に合わせて変更する)
#!/usr/local/bin/perl

# 配列の添字を0から始める
$[=0;

# 通信とファイルの処理をまとめたcommon.plを読み込む
require "common.pl";

# ランキングファイル
$RANKING_FILE="ranking.txt";

# ランキングを記録する個数
$RANKING_COUNT=5;

# 名前の最大文字数
$MAX_NAME_LENGTH=20;

# レスポンスのヘッダを出力する
print "Content-type: text/plain\n\n";

# パラメータの取得
@param=get_parameter();

# 名前とスコアを取り出す
$new_name=shift(@param);
$new_score=shift(@param);

# 名前とスコアに使われている文字の種類と
# 名前の文字数をチェックする
if (
  $new_name!~/[A-Za-z0-9]+/ ||
  length($new_name)>$MAX_NAME_LENGTH ||
  $new_score!~/[0-9]+/
) {
  # 不適切ならばエラーを出力して終了する
  print "error,parameter";
  exit;
}

# ランキングファイルの入力
@line=open_file($RANKING_FILE);

# スコアの登録
```

```
for ($i=0; $i<=$#line; $i++) {

  # ランキングファイルの各行から名前とスコアを取り出す
  @a=split(',', $line[$i]);
  $name=shift(@a);
  $score=shift(@a);

  # 新しいスコアが既存のスコアを上回っていたら、
  # 適切な位置に新しいスコアと名前を登録する
  if ($score<$new_score) {
    splice(@line, $i, 0, "$new_name,$new_score");
    last;
  }
}

# 規定の個数を超えた名前とスコアは削除する
splice(@line, $RANKING_COUNT);

# ランキングファイルの出力
close_file(@line);

# 正常に処理が終わったのでレスポンスとして「ok」を返す
print "ok\n";

# ランキングファイルの内容をレスポンスとして返す
for ($i=0; $i<=$#line; $i++) {
  print "$line[$i]\n";
}
```

CGIプログラムのテスト

サンプルのようなCGIプログラムを作ったら、JavaScriptプログラムを作る前に、単体でテストとデバッグを済ませておくのがお勧めです。CGIプログラムとJavaScriptプログラムのどちらに問題があっても正しく動作しないので、CGIプログラムの問題を先に解決しておけば、作業がとても楽になります。

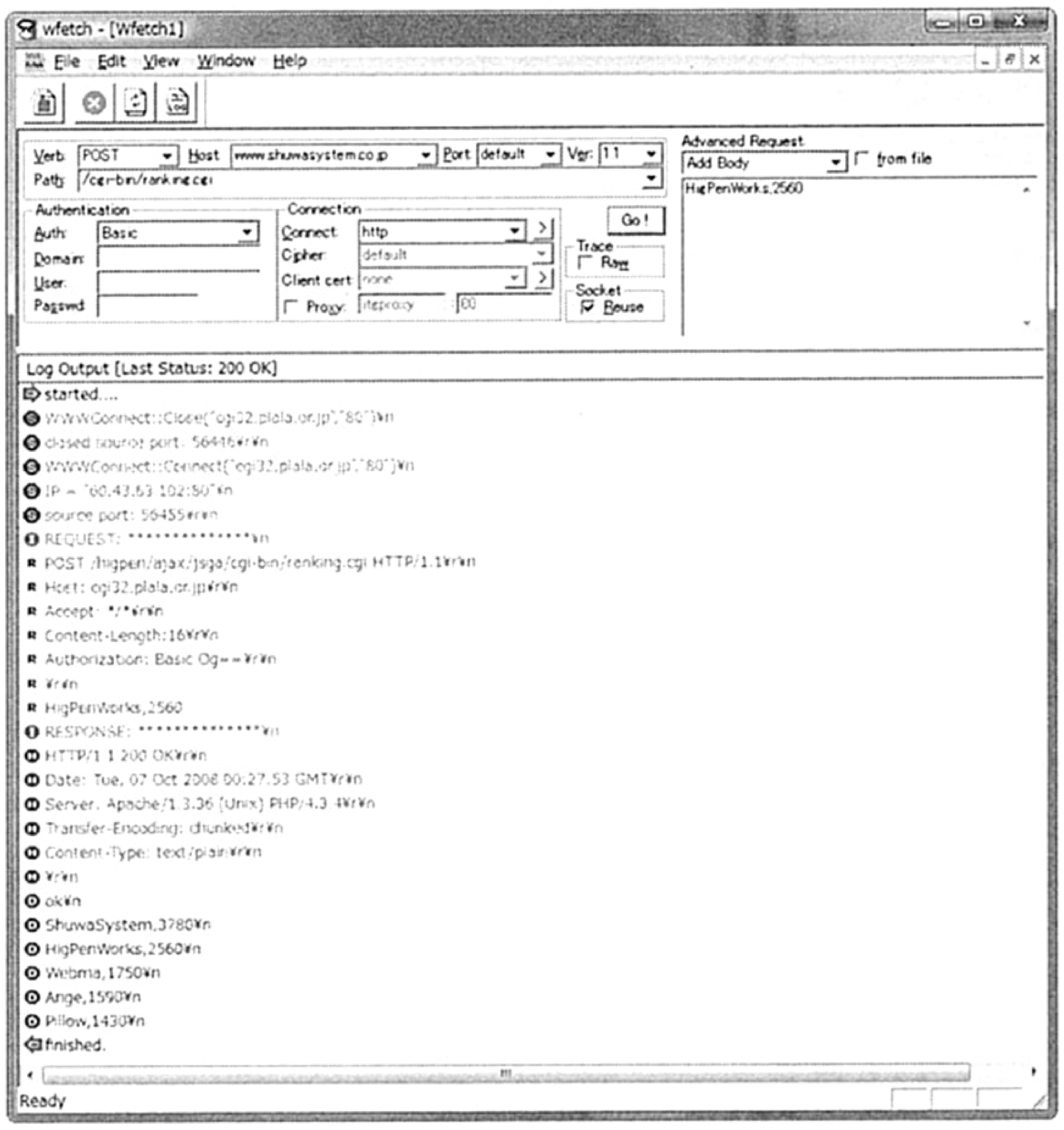
CGIプログラムをテストするために、本書ではwfetchと呼ばれるツールを使いました。wfetchはMicrosoftのWebサーバであるIIS(Internet Information Services)のリソースキットに収録されているツールの一つです。IISをインストールしなくても、wfetchだけで使うことができます。

wfetch(バージョン1.4)の入手先は以下の通りです。

http://www.microsoft.com/downloads/details.aspx?FamilyID=b134a806-d50e-4664-8348-da5c17129210&DisplayLang=en

wfetchを使うと、自由な内容のHTTPリクエストを送信し、レスポンスを受信して表示することができます(Fig.9)。ランキング機能のCGIプログラムをテストするには、以下のように設定します。

Fig.9: wfetch



- **Verb**
  HTTPリクエストの形式を表します。ここでは「POST」を指定します。
- **Host**
  ホスト(サーバ)を表します。動作環境に合わせて設定してください。例えば「www.shuwasystem.co.jp」のように入力します。
- **Path**
  サーバ上のファイルの位置を表します。動作環境に合わせて設定してください。例えば「/cgi-bin/ranking.cgi」のように入力します。
- **Advanced Request**
  リクエストパラメータの形式を表します。ここでは「Add Body」を指定します。

以上のように設定したら、Advanced Requestの大きな入力欄にリクエストパラメータを記述してから、Go!ボタンを押します。例えば、

```
HigPenWorks,2560
```

のようにリクエストパラメータを記述してボタンを押すと、wfetchがCGIプログラムに対してリクエストを送信し、CGIプログラムが返したレスポンスを表示します。

CGIプログラムが正しく動作すれば、ランキングの内容が表示されるはずです。リクエストパラメータにランキングよりも高いスコアを記述すれば、新しいスコアと名前がランキングに登録されます。

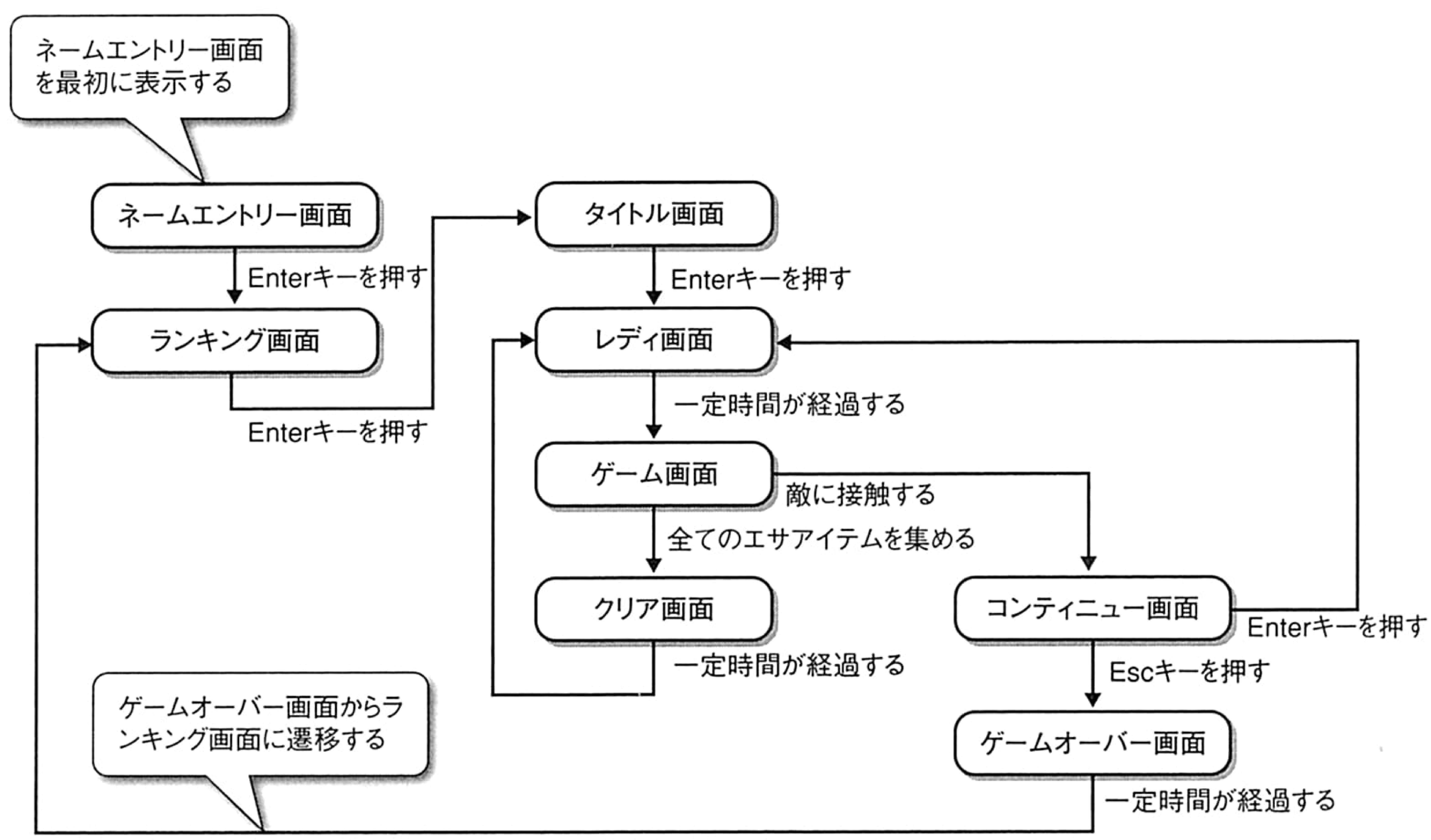
JavaScriptプログラムの制作

CGIプログラムが完成したら、JavaScriptプログラムの制作に移ります。第10章のプログラムに対して、以下のような機能を追加します。

- 起動時にネームエントリー画面を表示してプレイヤーの名前を入力させる
- CGIプログラムに名前とスコアをリクエストとして送信する
- CGIプログラムから受信したレスポンスを使って、ランキング画面を表示する

ネームエントリー画面とランキング画面を追加するので、第10章のプログラムとは画面の遷移が変わります(Fig.10)。起動時にはタイトル画面よりも先にランキング画面を表示しますが、これは起動してすぐにランキング画面を確認できるとテストに便利だったためなので、ネームエントリー画面からタイトル画面へ移るように変えてもよいでしょう。

▼ Fig.10: 画面の遷移



通信関連のJavaScriptプログラム

List4は通信関連のJavaScriptプログラムです。通信に関する処理はRequestオブジェクトにまとめました。

Requestオブジェクトは、XMLHttpRequestオブジェクトを使ってサーバにリクエストを送信し、レスポンスを受け取ります。最初にXMLHttpRequestオブジェクトを、

```
req=new XMLHttpRequest();
```

のように初期化します。ブラウザの種類によっては、

```
req=new ActiveXObject("Msxml2.XMLHTTP");
```

のように初期化する場合や、

```
req=new ActiveXObject("Microsoft.XMLHTTP");
```

のように初期化する場合もあるので、if文を使って場合分けをします。

XMLHttpRequestオブジェクトを使ってリクエストを送信するには、

```
req.open(method, uri, true);
```

のようにopen関数を使って、

- リクエストの種類(POSTなど)
- CGIプログラムのURI(/cgi-bin/ranking.cgiなど)
- 同期通信か非同期通信か(trueにすると非同期通信になる)

を指定します。非同期通信にすると、サーバからレスポンスが返るのを待つ間に、他の処理を進めておくことができます。

次に、

```
req.setRequestHeader("Content-Type", "text/plain");
```

のようにsetRequestHeader関数を使って、Content-typeエンティティヘッダフィールドを指定し

ます。ここではtext/plainを指定しました。
　ここまでの準備が終わったら、

```
req.send(message);
```

のようにsend関数を使って、リクエストを送信します。
　通信が終了したかどうかは、XMLHttpRequestオブジェクトのreadyStateプロパティで分かります。サンプルでは、

```
req.readyState==READYSTATE_COMPLETE
```

のような条件式で通信が終了したかどうかを調べます。一方、通信が成功したかどうかはstateプロパティで分かるので、

```
req.status==STATUS_OK
```

のような条件式で調べます。これらの条件式を使って、正常に通信が終了したときにtrueを返すisDone関数と、通信が失敗したときにtrueを返すisError関数を定義しました。
　レスポンスの内容はresponseTextプロパティに入ります。サンプルのgetResponse関数は、

```
return obj.xmlHttpRequest.responseText;
```

のように、レスポンスの文字列を返します。

List4: 通信関連のJavaScriptプログラム(common.js)

```
// XMLHttpRequestオブジェクトの通信状態
var READYSTATE_UNINI=0;
var READYSTATE_LOADING=1;
var READYSTATE_LOADED=2;
var READYSTATE_INTERACTIVE=3;
var READYSTATE_COMPLETE=4;

// レスポンスの状態
var STATUS_NOT_FOUND=404;
var STATUS_NOT_MODIFIED=304;
var STATUS_OK=200;

// リクエスト
function Request(){
```

```
var obj=this;

// XMLHttpRequestオブジェクトの生成
var req=false;
{
  // XMLHttpRequestを使用する場合
  if (window.XMLHttpRequest) {
    try {
      req=new XMLHttpRequest();
    } catch (e) {}
  } else

  // ActiveXObjectを使用する場合(IE/Windows)
  if (window.ActiveXObject) {
    try {
      req=new ActiveXObject("Msxml2.XMLHTTP");
    } catch (e) {
      try {
        req=new ActiveXObject("Microsoft.XMLHTTP");
      } catch(e) {}
    }
  }
}
obj.xmlHttpRequest=req;

// リクエストの送信
obj.send=function(method, uri, message) {
  var req=obj.xmlHttpRequest;

  // 送信ができない状態のときには、送信しない
  if (
    req.readyState!=READYSTATE_COMPLETE &&
    req.readyState!=READYSTATE_UNINI
  ) return;

  // リクエストを開く
  req.open(method, uri, true);

  // ヘッダの設定
  req.setRequestHeader("Content-Type", "text/plain");

  // リクエストの送信
  req.send(message);
}

// GETリクエストの送信
obj.doGet=function(uri) {
  obj.send("GET", uri, "");
```

```
  }

  // POSTリクエストの送信
  obj.doPost=function(uri, message) {
    obj.send("POST", uri, message);
  }

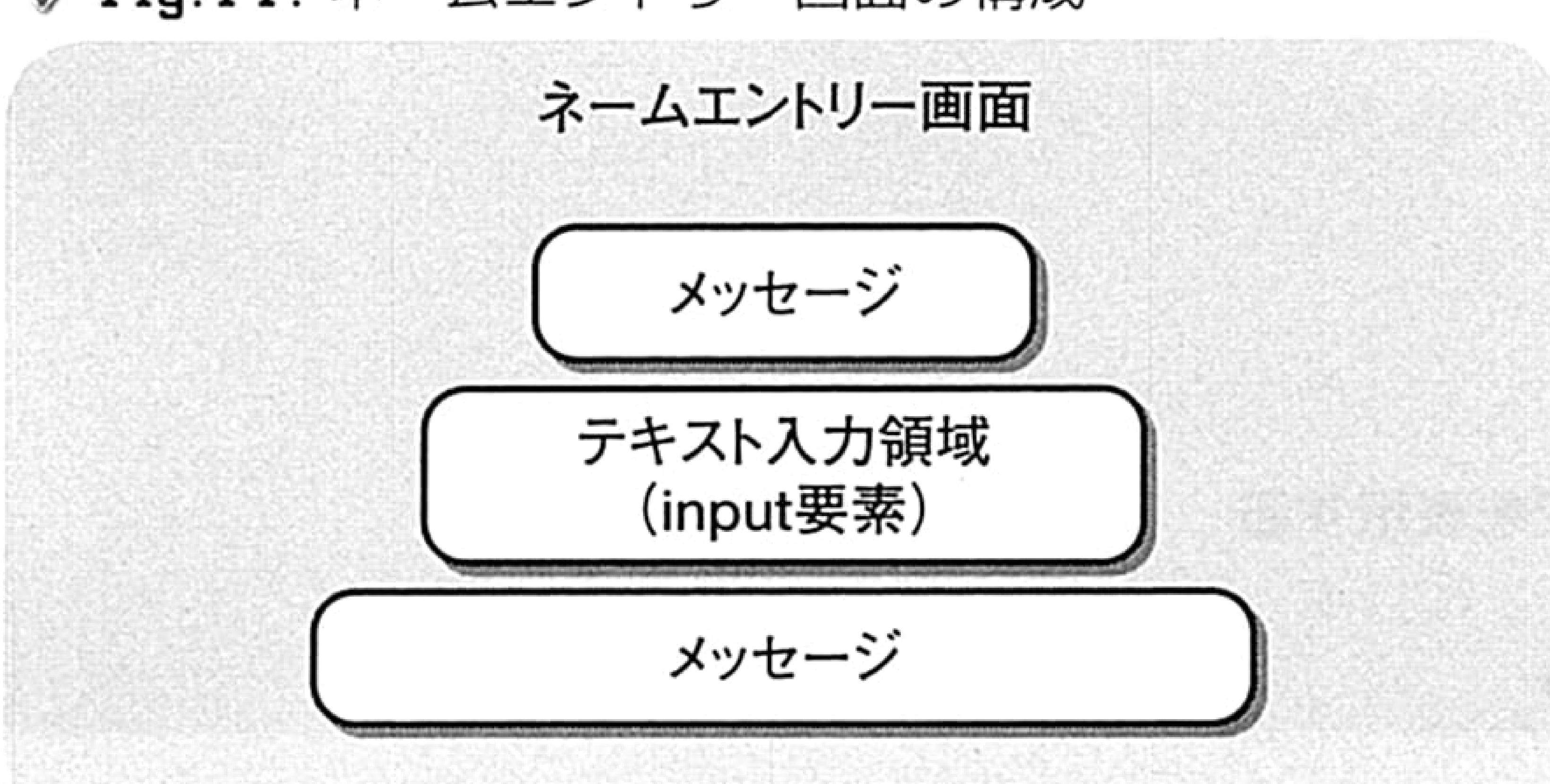
  // 状態の取得(通信成功・通信失敗)
  obj.isDone=function() {
    var req=obj.xmlHttpRequest;
    return (
      req.readyState==READYSTATE_COMPLETE &&
      req.status==STATUS_OK
    );
  }
  obj.isError=function() {
    var req=obj.xmlHttpRequest;
    return (
      req.readyState==READYSTATE_COMPLETE &&
      req.status!=STATUS_OK
    );
  }

  // レスポンスの取得
  obj.getResponse=function() {
    return obj.xmlHttpRequest.responseText;
  }
}
```

ネームエントリー画面のプログラム

サンプルでは起動時にネームエントリー画面を表示して、プレイヤーにランキング登録用の名前を入力させます。ここではHTMLのinput要素を使って、入力用の画面を作りました(Fig.11)。

▼ Fig.11: ネームエントリー画面の構成



List5はネームエントリー画面のプログラムです。ネームエントリー画面を初期化するinitNameEntry関数と、更新を行うupdateNameEntry関数に分かれています。initNameEntry関数は最初の一回だけ呼び出し、updateNameEntry関数は一定時間ごとに呼び出します。

初期化処理では、

```
<input id="nameInput">
```

のようなタグを使って、名前の入力領域を作成します。ここではidをnameInputに設定しました。HTML要素のidを設定しておくと、

```
obj.nameInput=document.getElementById("nameInput");
```

のように、getElementId関数を使って要素のオブジェクトを取得することができます。サンプルでは取得したオブジェクトを使って、フォントの種類やサイズ、テキストの位置などを設定します。

更新処理では、

```
return obj.rankingName.match(/^[A-Za-z0-9]+$/);
```

のように、入力した名前の形式を調べます。match関数は正規表現によるマッチングを行います。名前が英字(A～Z, a～z)と数字(0～9)だけで書かれている場合には、trueを返します。

List5: ネームエントリー画面(common.js)

```
// ゲーム本体
function Game() {
  // ...(中略)...

  // プレイヤーの名前
  obj.rankingName="";

  // ネームエントリー画面の初期化
  obj.initNameEntry=function() {

    // 画面の消去
    obj.clear();

    // 名前の入力領域とメッセージを表示する
    obj.divField.innerHTML=
      "<br><br><br><br><center>"+
```

```
        "Input Your Name<br><br>"+
        '<input id="nameInput"><br><br>'+
        "Press [Enter] Key To Submit<br>"+
        "</center>";

      // フォントの種類とサイズを設定する
      obj.divField.style.fontFamily=SCORE_FONT_FAMILY;
      obj.divField.style.fontSize=SCORE_FONT_SIZE;

      // 入力領域の形式・最大入力文字数・初期値を設定する
      obj.nameInput=document.getElementById("nameInput");
      obj.nameInput.type="text";
      obj.nameInput.maxlength=20;
      obj.nameInput.value="";

      // 入力領域のフォントの種類とサイズを設定する
      obj.nameInput.style.fontFamily=SCORE_FONT_FAMILY;
      obj.nameInput.style.fontSize=SCORE_FONT_SIZE;

      // 入力領域のテキストを中央寄せにする
      obj.nameInput.style.textAlign="center";

      // 入力領域にフォーカスを合わせる
      obj.nameInput.focus();

      // プレイヤーの名前を空にする
      obj.rankingName="";
    }

    // ネームエントリー画面の更新
    obj.updateNameEntry=function() {

      // 入力領域のテキストをプレイヤーの名前にする
      obj.rankingName=obj.nameInput.value;

      // プレイヤーの名前が適切な文字列ならばtrueを返す
      return obj.rankingName.match(/^[A-Za-z0-9]+$/);
    }
  }
```

ランキング画面のプログラム

ランキング画面はサーバから取得したランキング情報を表示します。表示にはHTMLのテーブル要素を使いました(Fig.12)。

▼ Fig.12: ランキング画面の構成



List6はランキング画面のプログラムです。ランキング画面を初期化するinitRanking関数と、更新するupdateRanking関数があります。initRanking関数は最初の一回だけ呼び出し、updateRanking関数は一定時間ごとに呼び出します。また、ランキング用のスコアを記録するために、setScore関数(p.129)も少し変更します。

setScore関数には、

```
if (obj.score%10==0) {
    obj.rankingScore=obj.score;
}
```

のような処理を追加します。ここではスコアの下一桁が0のとき、つまりコンティニューをしていないときに、現在のスコアをランキング用のスコアとして、rankingScoreプロパティに記録します。

initRanking関数では、通信中のメッセージを表示してから、

```
obj.rankingRequest.doPost(
    uri, obj.rankingName+","+obj.rankingScore
);
```

のようにリクエストを送信します。リクエストの内容は、

```
HigPenWorks,2560
```

のように、ネームエントリー画面で入力した名前と、ランキング用のスコアをカンマ(,)で区切った文字列です。

updateRanking関数では、

```
if (obj.rankingRequest.isDone()) {
```

のように、レスポンスが返ってきたかどうかを調べます。そして、

```
res=obj.rankingRequest.getResponse().split("\n");
```

のようにレスポンスを行単位で分割し、

```
if (res.shift()=="ok") {
```

として、最初の行がokかどうかを調べます。okならば、CGIプログラムの処理が正しく完了したということです。

正常終了した場合には、取得したランキングを表示します。ランキングの各行を、

```
a=res[i].split(",");
```

のように名前とスコアに分けてから、

```
s+=
    "<tr><td>"+(i+1)+"</td>"+
    "<td>"+a[0]+"</td>"+
    "<td>"+a[1]+"</td></tr>";
```

のように、HTMLのテーブルを使って表示します。trはテーブルの行を表す要素で、tdは項目を表す要素です。このほか、テーブルの項目名を表すth要素も使います。

▼ **List6**: ランキング画面(common.js)

```
// ゲーム本体
function Game() {
  // ...(中略)...

  // スコアの設定
  obj.setScore=function(score) {
    // ...(中略)...

    // まだコンティニューをしていないときには
    // スコアをランキング登録用のスコアにする
    if (obj.score%10==0) {
      obj.rankingScore=obj.score;
    }
  }

  // ランキング登録用のスコア
  obj.rankingScore=0;

  // リクエスト
  obj.rankingRequest=new Request();

  // ランキング画面の初期化
  obj.initRanking=function(uri) {

    // 画面の消去
    obj.clear();

    // 通信中のメッセージを表示する
    obj.divField.innerHTML=
      "<br><br><br><br><br><br><center>"+
      "Connecting To Server..."
      "</center>";

    // リクエストを送信する
    obj.rankingRequest.doPost(
      uri, obj.rankingName+","+obj.rankingScore
    );
  }

  // ランキング画面の更新
  obj.updateRanking=function() {
    var res, s, i, a;

    // 通信に成功したとき
    if (obj.rankingRequest.isDone()) {

      // レスポンスを行単位に分ける
```

```
      res=obj.rankingRequest.getResponse().split("\n");

      // 最初の行がokのとき
      if (res.shift()=="ok") {

        // ランキングを表示する
        s=
          "<br><br><center>"+
          "Top Players<br><br>"+
          '<table id="rankingTable">'+
          "<tr><th>Rank</th><th>Name</th>"+
          "<th>Score</th></tr>";
        for (i=0; i<res.length-1; i++) {

          // 名前とスコアを分けて表示する
          a=res[i].split(",");
          s+=
            "<tr><td>"+(i+1)+"</td>"+
            "<td>"+a[0]+"</td>"+
            "<td>"+a[1]+"</td></tr>";

          // 最上位のスコアをハイスコアにする
          if (i==0) {
            obj.setHighScore(a[1]);
          }
        }
        s+=
          "</table><br>"+
          "Press [Enter] Key To Proceed<br>"+
          "</center>";
        obj.divField.innerHTML=s;

        // フォントの種類とサイズ、テーブルの幅を設定する
        obj.rankingTable=document.getElementById("rankingTable");
        obj.rankingTable.style.fontFamily=SCORE_FONT_FAMILY;
        obj.rankingTable.style.fontSize=SCORE_FONT_SIZE;
        obj.rankingTable.style.width=FIELD_WIDTH/2;
      } else

      // 最初の行がokではないときは、再びサーバと通信する
      {
        obj.initRanking();
      }
    }

    // 通信に失敗したときは、再びサーバと通信する
    if (obj.rankingRequest.isError()) {
      obj.initRanking();
```

```
    }
  }
}
```

ゲーム本体のプログラム

List7はゲーム本体のプログラムです。第10章のプログラムに対して、ネームエントリー画面とランキング画面を追加しました。

新しい画面の遷移を処理するために、gameオブジェクトのonUpdate関数を変更します。ネームエントリー画面については、

- GS_NAME : ネームエントリー画面の初期化を行う
- GS_NAME2 : プレイヤーが名前とEnterキーを入力するのを待つ
- GS_NAME3 : プレイヤーがEnterキーを放すのを待つ

という3つの状態を作りました。最後にEnterキーを放すのを待つのは、次に遷移するランキング画面でEnterキーが押されていると、ランキングが表示される前に別の画面に遷移してしまうからです。

ランキング画面についても同様に、

- GS_RANKING : 通信中のメッセージを表示し、リクエストを送信する
- GS_RANKING2 : レスポンスを受信し、ランキングを表示して、プレイヤーがEnterキーを押すのを待つ
- GS_RANKING3 : プレイヤーがEnterキーを放すのを待つ

のような3つの状態を作りました。最初にリクエストを送信し、別の状態でレスポンスを受信します。レスポンスを受信するまでには通信中のメッセージを表示しますが、ここでもっと凝った処理をすることもできます。例えばアニメーションを再生することも可能です。あるいは通信中にゲームを進行することによって、本格的なオンラインゲームを実現することもできるでしょう。これは非同期通信ならではの利点です。

▼ **List7**: ゲームの本体(ranking.js)

```
// 現在の画面を表す定数(ネームエントリー・ランキング)
var GS_NAME=11;
var GS_NAME2=12;
var GS_NAME3=13;
var GS_RANKING=14;
var GS_RANKING2=15;
```

```
var GS_RANKING3=16;

// メインルーチン
function main() {
  // ...(中略)...

  // 更新処理
  game.onUpdate=function() {

    // 画面に応じて分岐する
    switch (game.state) {

      // ネームエントリー画面の前処理
      case GS_NAME:

        // ネームエントリー画面を初期化する
        game.initNameEntry();

        // ネームエントリー画面の本体に遷移する
        game.state=GS_NAME2;

      // ネームエントリー画面の本体
      case GS_NAME2:

        // 正しい形式の名前を入力して
        // Enterキーを押したとき
        if (
          game.updateNameEntry() &&
          game.isKey(KEY_ENTER)
        ) {
          // ネームエントリー画面の後処理に遷移する
          game.state=GS_NAME3;
        }
        break;

      // ネームエントリー画面の後処理
      case GS_NAME3:

        // Enterキーを放したとき
        if (!game.isKey(KEY_ENTER)) {

          // ランキング画面に遷移する
          game.state=GS_RANKING;
        }
        break;

      // ゲームオーバー画面の本体
      case GS_GAMEOVER2:
```

```
        // タイマーの更新
        game.time++;

        // タイマーが一定値に達したときの処理
        if (game.time==20) {

          // 画像の消去
          initSplash("void.png");

          // ランキング画面に遷移する
          game.state=GS_RANKING;
        }
        break;

      // ランキング画面の前処理
      case GS_RANKING:

        // ランキング画面を初期化する
        game.initRanking("cgi-bin/ranking.cgi");

        // ランキング画面の本体に遷移する
        game.state=GS_RANKING2;

      // ランキング画面の本体
      case GS_RANKING2:

        // ランキング画面を更新する
        game.updateRanking();

        // Enterキーを押したらランキング画面の後処理に遷移する
        if (game.isKey(KEY_ENTER)) {
          game.state=GS_RANKING3;
        }
        break;

      // ランキング画面の後処理
      case GS_RANKING3:

        // Enterキーを放したらタイトル画面に遷移する
        if (!game.isKey(KEY_ENTER)) {
          game.state=GS_TITLE;
        }
        break;
    }
  }
}
```

おわりに

ゲームを遊んでいると、誰しも一度くらいは自分でゲームを作ってみたいと思うものです。このゲームのここをこうアレンジしたらもっと面白くなるとか、今までに全くなかった斬新なアイディアが浮かんできたのでゲームに仕上げてみたいといった考えが、ゲームをプレイしていると浮かんできます。

最近のゲームは絵や音には非常に凝っていますが、ゲームのロジックは意外に単純で、昔からあるゲームとさほど変わりません。ここまで本書をお読みいただいた方ならば、絵や音のようなコンテンツ作りに手間をかけさえすれば、本格的なゲームを作るのも決して難しくはないと感じていただけるでしょう。

ゲーム制作に最初の一歩を踏み出すときはなかなか大変そうに思いますが、一度基本を学んでしまえば、あとはとてもスムーズに進んでいきます。本書を読み進むうちに、きっと色々なゲームのアイディアが湧いてきたことでしょう。そのアイディアをゲームという形に仕上げて、ぜひ多くの人を楽しませてください。

皆さんのゲームにいつかどこかで出会えることを、心から願っています。

索　引

数字・記号

A

B

C

D

E

F

G

H

I

J

V

W

X

Z

あ

か

さ

た

な

は

ま

や

ら

著者紹介

◆松浦　健一郎（まつうら　けんいちろう）

東京大学工学系研究科電子工学専攻修士課程を修了後、研究所勤務を経て、現在は趣味と実益を兼ねつつフリーのプログラマ＆ライターとして活動中。関心と仕事の範囲はプログラミングを中心にコンピュータ全般に及ぶが、最も興味がある分野はプログラミング言語作りとゲーム作り。

◆司　ゆき（つかさ　ゆき）

東京大学理学系研究科情報科学専攻修士課程修了。学部生時代からライターおよびプログラマの仕事を始める。書籍の執筆のほか、ソフトウェアの受注開発も行う。

著者Webサイト「ひぐぺん工房」

http://cgi32.plala.or.jp/higpen/gate.shtml

著書（いずれも共著）

「Windows Vistaガジェットプログラミング」
「うぇぶマスコットをつくろう!!－for Ajax」
（以上、秀和システム刊）

「はじめてのJBuilder4」
「Delphi DB&Webプログラミング」
「はじめてのJBuilder6」
「デスクトップマスコットを作ろう!!」
「シューティングゲーム アルゴリズム マニアックス」
「ゲームエフェクト マニアックス」
「Javaのココロ」
「シューティングゲーム プログラミング」
「アクションゲーム アルゴリズム マニアックス」
「3D格闘ゲーム プログラミング」
「Visual C++ 2005 実用サンプルプログラム Windowsプログラミング Tips 108」
「パズルゲーム アルゴリズム マニアックス」
（以上、ソフトバンククリエイティブ刊）

本書サポートページ

本書で使われるサンプルコードは秀和システムのウェブページからダウンロードして学べます。

■秀和システムのウェブサイト
http://www.shuwasystem.co.jp/

■本書サポートページ
http://www.shuwasystem.co.jp/support/7980html/2128.html

ゲームアルゴリズムレシピ for JavaScript（フォー ジャバスクリプト）

発行日　2008年　12月　1日　　第1版第1刷

著　者　松浦（まつうら）　健一郎（けんいちろう）／司（つかさ）　ゆき

発行者　斉藤　和邦

発行所　株式会社　秀和システム
〒107-0062　東京都港区南青山1-26-1 寿光ビル5F
Tel 03-3470-4947（販売）
Fax 03-3405-7538

印刷所　株式会社シナノ

ISBN978-4-7980-2128-7 C3055

定価はカバーに表示してあります。
乱丁本・落丁本はお取りかえいたします。
本書に関するご質問については、ご質問の内容と住所、氏名、電話番号を明記のうえ、当社編集部宛FAXまたは書面にてお送りください。お電話によるご質問は受け付けておりませんのであらかじめご了承ください。

ISBN978-4-7980-2128-7
C3055 ¥2800E
9784798021287
1923055028005
定価　(本体 2800円+税)

Game Algorithm Recipe

秀和システム